What's New

- **HashiCorp Vault Support for Database Credentials**

  TIBCO GridServer® now integrates with HashiCorp Vault to securely store and retrieve database credentials for the reporting database. This allows you to centralize credential management, enhance security, and maintain compliance with enterprise standards. You can use Token-based and AppRole-based Vault authentication methods.

  For more information, see "Configuring HashiCorp Vault for Reporting Database Credentials" section in TIBCO GridServer® Installation guide .

- **Active Directory Authentication for the Microsoft SQL Server Reporting Database**

  TIBCO GridServer® now supports on-premises Active Directory (AD) authentication for the Microsoft SQL Server reporting database as an alternative to standard SQL Server authentication. This feature is supported on both Windows and Linux operating systems running the Manager. This enhancement allows you to configure Windows Integrated Authentication for your reporting database, meeting stringent security requirements and ensuring seamless integration with your existing AD infrastructure.

  For more information, see "Configuring Active Directory (AD) Authentication for the SQL Server Reporting Database (On-Premise)" section in theTIBCO GridServer® Installation guide.

- **Apply License Using Script**

  TIBCO GridServer® supports applying a license file through a script. This eliminates the need to log in to the Director UI for license installation.

  During a silent installation, set the new `licenseFile` property in `install.silent`. Provide the full path to your license file. The system applies the license automatically as part of the install. For more information, refer to "Silent Installation" section in the TIBCO GridServer® Installation guide.

- **Apache Tomcat® 10.1.55 Upgrade**

  The embedded Apache Tomcat server has been upgraded to version 10.1.55. As Tomcat 10.1 implements Jakarta EE 9 or later, the GridServer Manager web application has migrated from the `javax.*` namespace to the `jakarta.*` namespace. As a result, the REST and SOAP API connectors and authenticators have also been updated to support the new namespace.

Installation Guide

- Typographical Conventions
- Installation Overview
  - Preparing for Installation
  - Know_the_Basics
  - GridServer System Requirements
  - Optimize GridServer Manager Architecture
    - About GridServer Manager
    - Redundancy
    - Grid Size
    - Component OS and Version Interoperability
    - Other Considerations
  - Configure Your Network
    - Name Service
    - Director-Broker Communication
    - Manager-Engine and Driver-Manager Communication
    - Configuring the Manager by Using NAT
    - Using a VPN
    - Resource Synchronization
    - Direct Data Transfer
    - SSL
  - Install the GridServer Components
    - GridServer Manager
    - GridServer Engine
    - GridServer Driver
    - SpeedLink

        - Database
- Manager Installation
    - Installing DataSynapse GridServer
        - Task A: Copy Files Before Installation
        - Task B: Configure Server Settings
        - Task C: Start the Manager
        - Task D: Initialize Your Manager
    - Silent Installation
    - Configuration Options
        - Changing the HTTP Ports
    - A Sample UNIX rc.d Script
    - A Sample Systemd Unit File
    - Authentication of Grid Components
- Windows Engine Installation
    - Installing Windows Engines
    - Command Line Unattended Install
    - Network Install
    - Engine JRE Synchronization
    - Engine Permissions
    - Configuring Permissions for Processor Utilization Mode
    - Configuring Run-As for Windows Engines
    - Engine Configuration and Management
- UNIX Engine Installation
    - Installing_Unix_Engines
    - Engine JRE Synchronization
    - Configuring Run-As for UNIX Engines
    - Running the Engine

    - Engine Configuration and Management
- Enabling Engine Daemon Authentication
- Verify Installation
    - Submitting a Service Test
    - Monitoring a Service Test
    - Troubleshooting
- Uninstalling GridServer
    - Uninstalling GridServer on Windows Systems
        - Uninstalling an Engine
        - Manual Uninstall
        - Uninstalling a Manager
    - Uninstalling GridServer on UNIX Systems
    - Uninstalling a Service Pack
- Upgrading GridServer
    - Requirements Changes
    - GridServer 7.1.1 Upgrade Checklist
        - Prior to Upgrade
        - After Stopping the Manager
        - During Installation
        - After Installation
    - Mixed-Version Deployments
    - Rolling Upgrades
        - Prerequisite Actions
        - Upgrading a Grid
    - Installing Hotfixes
- Securing Passwords and Internal.script
- Configuring SSL

    - About SSL
    - Communication Overview
    - Certificate Overview
    - Key and Certificate Location and Generation
    - Manager HTTPS Configuration
        - Configuring HTTPS
    - Driver HTTPS
    - Engines and Engine Daemon HTTPS
    - Broker and Director SSL
    - SSL On Large Grids
- Configuring a Reporting Database
    - Reporting Database Requirements
    - Configuring the Reporting Database
    - Configuring Active Directory (AD) Authentication for the SQL Server Reporting Database (On-Premise)
    - Configuring HashiCorp Vault for Reporting Database Credentials
- Deploying With Openshift
    - Introduction
        - About OpenShift
        - About Kubernetes
        - About the GridServer Containerization
    - Deploying the GridServer Engine Dockerfile into OpenShift
- Configuring Scheduler Instrumentation
- Configuring Engine Instrumentation

Typographical Conventions

The following table lists the typographical conventions used in this guide:

CONVENTION	USE
`TIBCO_HOME`	Many TIBCO products must be installed within the same home directory. This directory is referenced in the documentation as `TIBCO_HOME`. The default value of `TIBCO_HOME` depends on the operating system. For example, on Windows systems, the default value is `C:\tibco`.
`DS_INSTALL`	TIBCO GridServer® installs into a directory within `TIBCO_HOME` named datasynapse. This directory is referenced in the documentation as `DS_INSTALL`. The default value of `DS_INSTALL` depends on the operating system. For example, on Windows systems, the default installation directory is `C:\tibco\datasynapse`.
`DS_MANAGER`	The *Manager directory* contains the read-only software files; by default, it is a directory within `DS_INSTALL` named `manager`, and is referred to as `DS_MANAGER`. For example, on Windows systems, the default Manager directory is `C:\tibco\datasynapse\manager`.
`DS_DATA`	The *data directory* is the location of all volatile files used by the application server such as server properties and configuration. By default, it is a directory within `DS_INSTALL` named `manager-data`, and is referred to as `DS_DATA`. For example, on Windows systems, the default data directory is `C:\tibco\datasynapse\manager-data`.

Installation Overview

This is your starting point for planning, installing, and configuring TIBCO GridServer® for your site. This guide describes how to install, test, and uninstall the Manager, Drivers, and Engines on Windows and UNIX platforms.

This guide is for a DataSynapse GridServer Manager or technology manager planning or assessing requirements for a GridServer installation and an engineer or administrator installing and configuring a GridServer Manager system.

Preparing for Installation

This section lists prerequisites for a GridServer installation and outlines the installation steps. This section assists developers, system architects, and network architects in planning a GridServer production deployment. Read this section carefully to prepare for installation.

Before you begin your GridServer installation, determine what GridServer components you must install and how to configure your overall system. You must also prepare for the installation, meet system requirements, and have an overall understanding of GridServer and the installation process.

Know the Basics

This guide assumes that you know GridServer concepts. If you do not, see the TIBCO GridServer® Introducing TIBCO GridServer® for information about the GridServer component architecture and principles of operation.

You must also be familiar with Windows and UNIX operating systems and TCP/IP networking.

GridServer System Requirements

Ensure that your environment meets the minimum hardware and software system requirements for memory, disk space, and processor speed. See the readme file included in the product archive for the complete list of all hardware and software requirements, including supported operating systems, Java or .NET software, web browsers, and accompanying components. Note that depending on the applications you virtualize, you might have to exceed these requirements accordingly.

Optimize GridServer Manager Architecture

You can deploy the GridServer Manager architecture to give varying degrees of redundancy and load sharing, depending on the available computing resources. To plan an architecture that best supports your needs:

- Determine how your facility wants to use GridServer.
- Estimate expected transaction volume and amount of work.
- Survey hardware and networking requirements for the installation.

About GridServer Manager

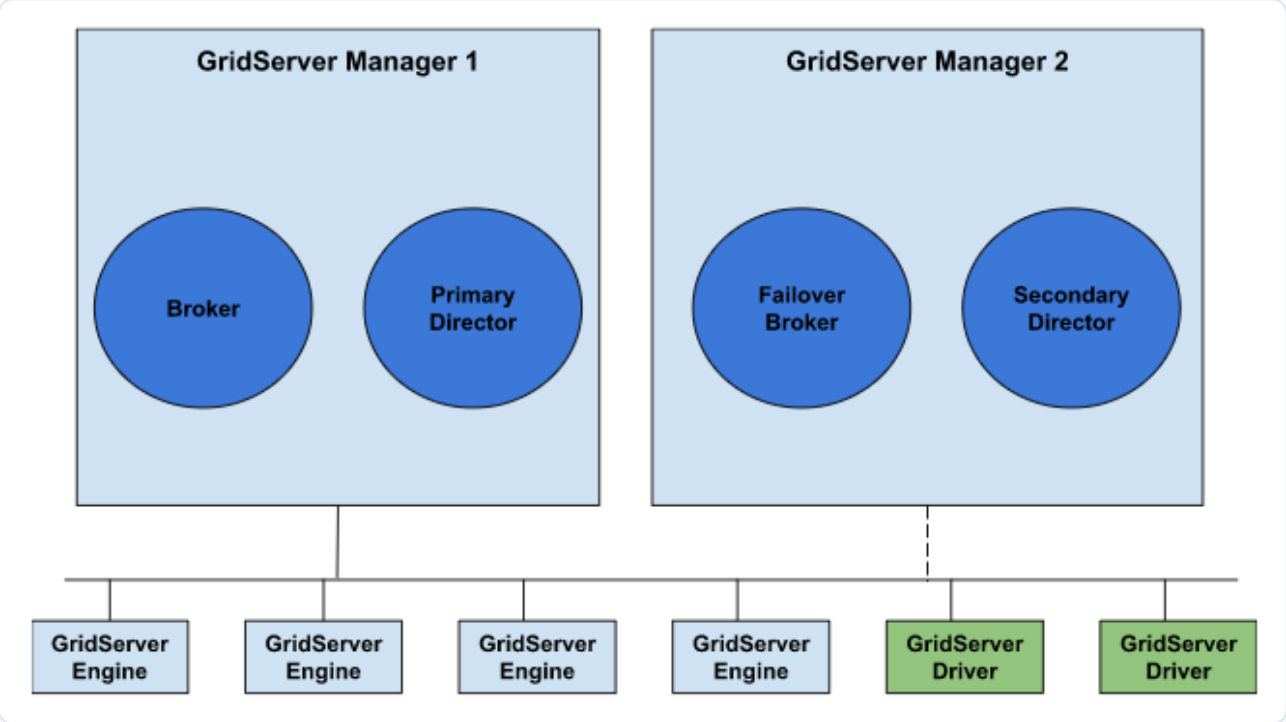
A minimal configuration of GridServer consists of a single Manager configured with a Primary Director and a single Broker. You can add additional Managers containing more Brokers or Directors to address redundancy, volume, or other considerations.

You set an initial configuration of the Manager at the time of installing the Manager, on the **Manager Type**screen. To change the configuration of a running Manager, go to **Admin > System Admin > Manager Reinstall**, follow the screens for Manager reinstallation, and restart the Manager when the configuration is complete.

Redundancy

If you have a minimal configuration including a single Director and single Broker, Engines and Drivers log in to the Director. In this configuration, if the Director fails (as in the case of hardware or network failure), Drivers or Engines cannot establish new connections.

To prevent this problem, run a second Manager with a Secondary Director, and configure Engines and Drivers with the address of both Directors. If the Primary Director fails, Engines and Drivers contact the Secondary Director, which routes Engines and Drivers to Brokers in the same manner as the Primary Director. The following figure shows an implementation with two Managers.



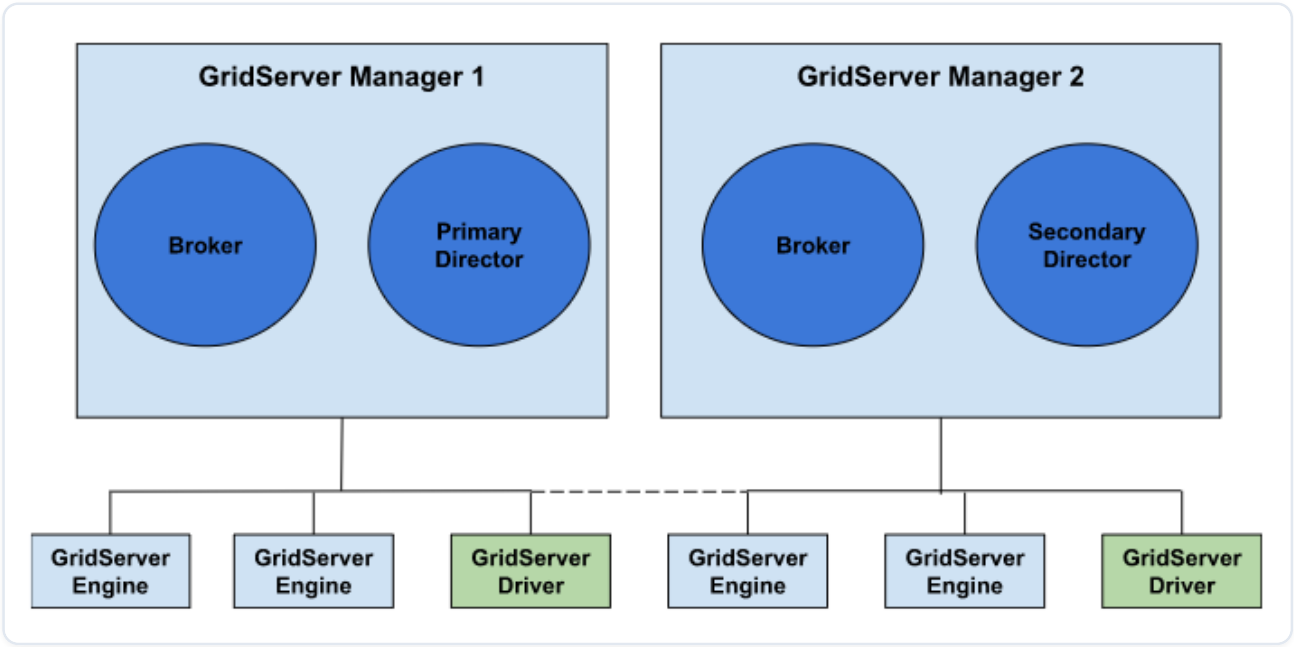
In this typical GridServer installation, there is a Primary Director and a Secondary Director installed on two separate systems. On the same system as the Primary Director, there is a Live Broker. A Failover Broker is installed on the same system as the Secondary Director. Add Engines and

Drivers to the cluster as desired. This configuration is fully redundant but does not do any load sharing.

In addition to redundant Directors, a Broker can have a backup on another Manager. You can designate a Broker to be a Failover Broker on a Manager during installation or reinstallation. If no other regular Brokers are available (because of a failure), Directors temporarily route Drivers and Engines to Failover Brokers. The figure to the left shows a Failover Broker on the second Manager.

For more information about redundancy, see the *TIBCO GridServer® Administration*.

Grid Size



This GridServer configuration shows a Primary Director and a Secondary Director on separate systems and live Brokers on both systems. You can add Engines and Drivers to the cluster as desired. This configuration is redundant.

In larger grids, the number of Engines in the grid can require more capability than a single Broker can provide. To distribute load, add more Brokers to other Managers at installation. For example, the figure to the right shows a two-Manager system with two Brokers. Directors route Engines and Drivers to Brokers in round-robin fashion.

Component OS and Version Interoperability

Running Brokers on the same OS as other components is not a requirement and gives no performance benefit. However, running the same OS for all components does simplify administration and troubleshooting.

A GridServer installation typically requires that GridServer components all reflect the same version. When you upgrade a GridServer Manager, it is best to upgrade all Drivers. Engines upgrade themselves automatically.

Other Considerations

Several other factors can influence how you integrate GridServer with your computing environment:

- Instead of using one grid for all types of Services, you might prefer to divide different subsets of Services (for example, by size or priority) to different Directors.
- Your network can dictate how to plan your Manager environment. For example, if you have offices in two parts of the country and a relatively slow extranet but a fast intranet in each location, you can install a Manager in each location.
- Different Managers can support data used for different Service types. For example, you can configure one Manager for Services accessing a SQL database and another Manager for Services that don’t access the database.

With this flexibility, it’s possible to plan a Manager model to provide a work space that facili-tates your workload and traffic. For more information about designing your Manager environment, contact our Integration Services staff, and we can help you determine how to best configure your installation.

Configure Your Network

Since GridServer is a distributed computing application, successful deployment depends on your network configuration. Treat GridServer Managers the same way you treat your other mission-critical file and application servers: assign GridServer Managers static IP addresses and resolvable DNS hostnames. You can configure GridServer Engines and Drivers in several different ways. To receive the full benefit of peer-to-peer communication you must enable communication between Engines and Drivers (the default), but you can also configure GridServer to work with a hub-and-spoke architecture by disabling Direct Data Transfer.

Name Service

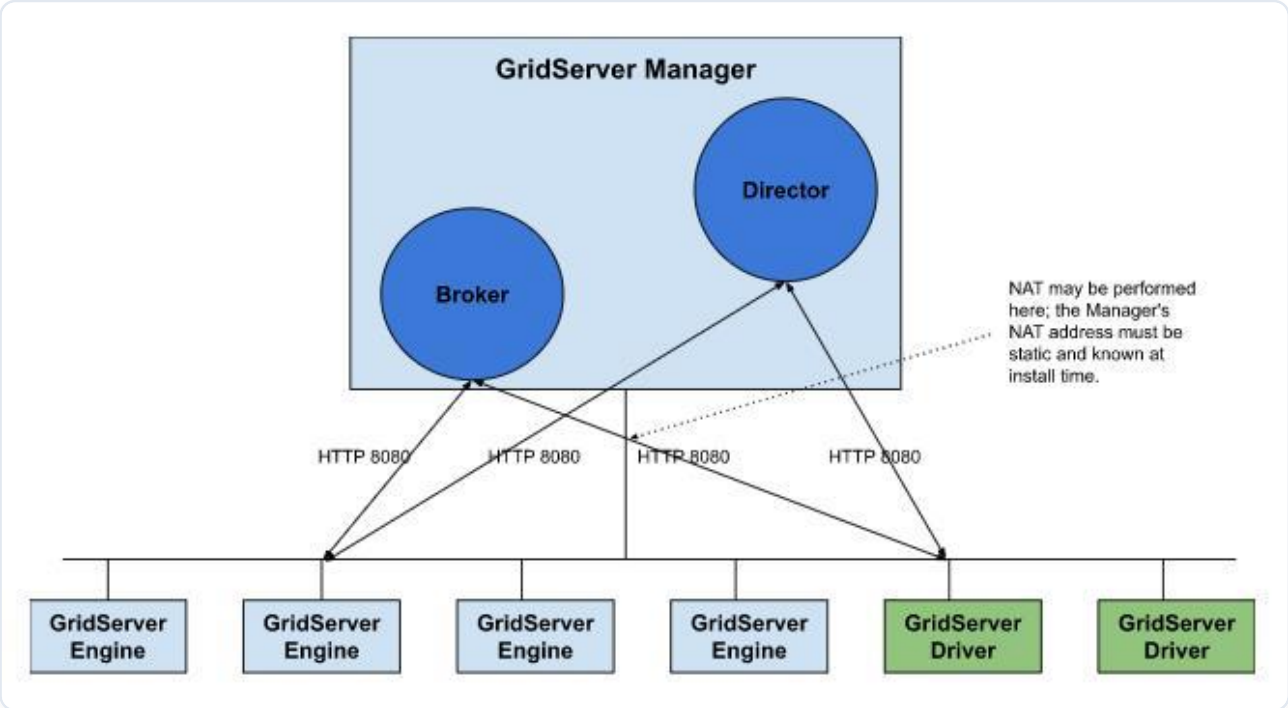
Run GridServer Managers on systems with static IP addresses and resolvable DNS hostnames. In a pure Windows environment, it is possible to run GridServer using just WINS name resolution, but this is not recommended for larger deployments or heterogeneous environments. GridServer does not support DHCP-assigned client addresses for Manager components. DHCP-assigned addresses are acceptable for Engines and Drivers.

Director-Broker Communication

Directors and Brokers communicate using TCP on port 5635. This port is used for both initial login and further communication. If you need to specify a different port (for example, if you are running multiple instances on a single machine), you can do so at installation, or by reinstalling the Manager at **Admin > System Admin > Manager Reinstall**.

Manager-Engine and Driver-Manager Communication

All communication between Engines and Managers (Directors and Brokers) and between Drivers and Managers uses the HTTP protocol. The Engine or Driver acts as HTTP client and the Manager acts as HTTP server.



All communication between Engines and Managers or Drivers and Managers is with the HTTP protocol on the port assigned when the Manager is installed. In this diagram we assume the assigned port is 8080. All HTTP communication is initiated by the Engine or Driver as the HTTP client.

An Engine Daemon periodically checks if its IP address has changed. If so, the Engine Daemon restarts itself and all Engine instances so that they use the new IP address.

Configuring the Manager by Using NAT

You can configure the Manager to work with a NAT device between the Manager and the Engines or Drivers. If the untranslated site of the network has no Engines, you can configure to work with a NAT device by specifying the external (translated) address of the NAT device when referring to the

Manager address in the Manager URL field during Manager installation, in Driver installation, and in Windows and Linux Engine installation.

If clients (Engines and Drivers) exist on both sides of a NAT device, you can configure NAT Translation on the Manager. This enables the URLs used for Manager-to-client communication.

Configure NAT Translation in the GridServer Administration Tool

1. Go to **Admin > System Admin > Manager Configuration > Communication**.
2. Find the **NAT Translation**heading. The first field, **NAT Translation Range**, specifies the range of IP addresses within the NAT untranslated network to be translated.
3. In the **NAT URL**field, type the external URL to use for the Manager on the translated side of the network. If you want the NAT URL used for clients inside the NAT Translation range, set **Translate Inside Range (True/False)**to **True**; to use NAT translation for clients outside the NAT Range, set this to **Outside**.

Using a VPN

Engines and Clients can run behind a VPN, provided that the Engine Daemon or Driver can determine the correct IP address to use. If VPN client software on the Engine/Client machine handles the VPN, you can possibly specify which IP address to use. For Engines, in the Engine Configuration, use the **Net Mask**setting to force the engine to pick the VPN interface. For Clients, you can set the `DSLocalIPAddress` property to the IP number of the correct interface.

If an external device like a router handles the Engine/Client-side VPN, it is unlikely that the Engine Daemon or Driver can determine the VPN address. Running Engines or Clients in this configuration is not possible.

Resource Synchronization

Normally, GridServer synchronizes resources from Directors to Brokers, then from Brokers to Engines. However, in situations when you want only to deploy resources to a subset of Engines, you can disable resource synchronization in the Engine Configuration, and then set the lib directory to point to a shared directory containing resources. On the Engine Configuration page, set the **Synchronize Resources**property in the **Resource Validation**section to false to disable resource synchronization for that Engine Configuration.

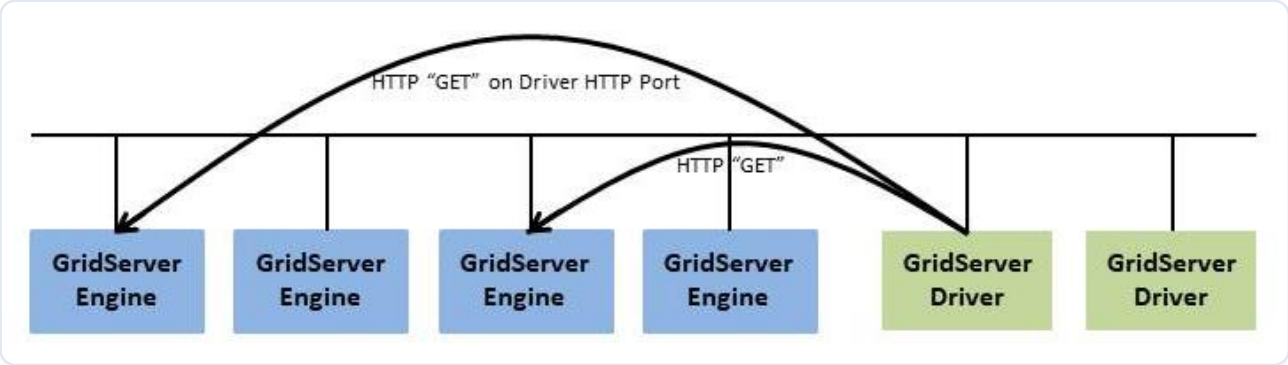
Consider the following business use case: using offsite rented systems (on-demand systems) for Engines. An offsite shared collocation facility hosts some Engine systems. You do not fully trust the colocation provider to erase all sensitive data before reusing the systems, so you keep all resources, including code, on a file server at your site. The offsite Engines have a special offsite

Engine Configuration pointing lib directories to a file server at your site. You configure other Engines at your site to use resource deployment in the normal way.

Direct Data Transfer

By default, GridServer uses Direct Data Transfer, or peer-to-peer communication, to optimize data throughput between Drivers and Engines. Without Direct Data Transfer, all task input and output goes through the Manager. Sending the input and output through the Manager uses more memory and disk on the Manager and results in lower throughput.

Using Direct Data Transfer, the Driver and Engine nodes do the “heavy lifting,” and only lightweight messages go through the Manager. Direct Data Transfer requires each peer to know the IP address that it presents to other peers. In most cases, therefore, Direct Data Transfer precludes the use of NAT between the peers. Likewise, Direct Data Transfer does not support proxies.

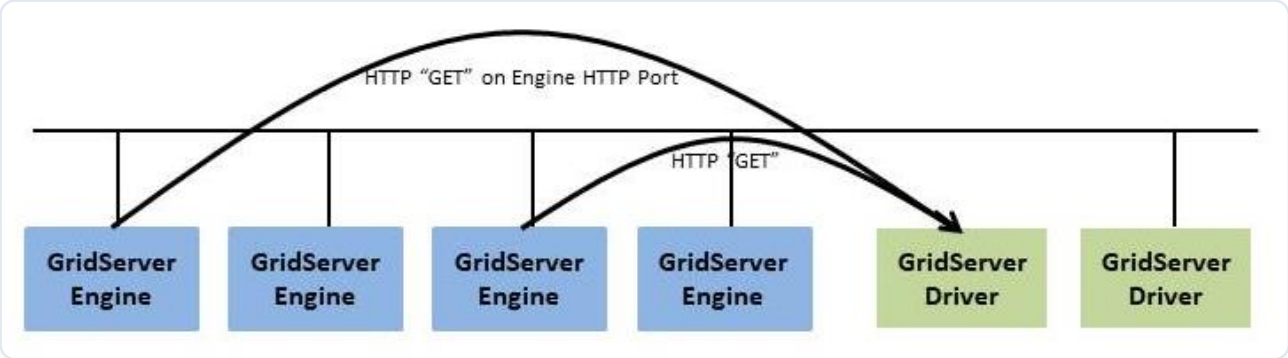


When using Direct Data Transfer to move TaskInput data from Drivers to Engines, the Engines pull the data via HTTP directly from the Driver. You can configure the Driver's embedded HTTP server to use a static TCP port or an ephemeral TCP port according to the `driver.properties` file.

You can also install an optional Engine Hook to enable communication between Drivers and Engines when NAT is in use.

For GridServer deployments that use NAT and do not have the optional Engine Hook, you can support NAT between Drivers and Engines by disabling peer-to-peer communication in one of two ways:

- If, from the perspective of the Drivers, the Engines are behind a NAT device, the Engines cannot provide peer-to-peer communication. In this case, disable Direct Data Transfer in the Engine configuration.
- If, from the perspective of the Engines, the Drivers are behind a NAT device, then the Drivers cannot provide peer-to-peer communication. Provide the Driver addresses if you know them in advance. Otherwise, disable Direct Data Transfer in the Driver properties file.



When using Direct Data Transfer to move TaskOutput data from Engines to Drivers, the Drivers pull the data via HTTP directly from the Engines.

SSL

Depending on your application server, you can configure your Manager to use SSL selectively or for all component communication and administration.

For information about using SSL, see Configuring SSL.

Install the GridServer Components

After preparing for your installation, install the GridServer components. This section introduces the components and directs you to their installation instructions.

GridServer Manager

To install a Manager, follow the installation procedure in the Manager Installation section. You must have a running Manager before you install an Engine.

For more information about installing Managers, see Installing DataSynapse GridServer.

GridServer Engine

GridServer Engines come in two variants: Windows Engines and UNIX Engines.

- The Windows Engine is packaged as an MSI installer. It can be manually installed.
- UNIX Engines are packaged as `tar.gz` archives.

Security Considerations

On the Windows platform, Engines and Managers run as services owned by a user specified during installation.

To prevent unauthorized users from accessing files in the Engine's directory tree, the Windows system administrator can set up the Engine directory to:

- Not inherit permission from the parent directory and
- Grant full access to the built-in `SYSTEM` and `SERVICE` users but not grant access to any other users.

On UNIX platforms, Engines and Managers install with the file permissions of the user that installs them. That same user must start the Engines and Managers. For this reason, choose a Manager port above 1024.

For directions on installing Engines, see Windows Engine Installation, UNIX Engine Installation, and Enabling Engine Daemon Authentication.

GridServer Driver

Application programs use the GridServer Driver to communicate with the GridServer Manager and thereby leverage the grid's compute resources. Your application deployment usually bundles the Driver deployment. For more information about installing the GridServer SDK, which includes all Drivers, see the *TIBCO GridServer® Developer's Guide*.

SpeedLink

SpeedLink is a high-throughput extension to GridServer, and is included in the GridServer SDK. For more information, see the *TIBCO GridServer® Speedlink.*.

Database

You can also configure GridServer to use an external reporting database, on an enterprise database system. For more information about database configuration, see Configuring a Reporting Database.

You can optionally have GridServer to retrieve the reporting database credentials from HashiCorp Vault instead of storing them in the Manager configuration. For details, see Configuring HashiCorp Vault for Reporting Database Credentials.

Manager Installation

To begin your DataSynapse GridServer installation, you must install one or more Managers. This section describes how to install the Manager and its components such as Primary and Secondary Directors, and Standalone and Failover Brokers.

This section assumes that you are familiar with the previous section, “Installation Overview” and know which components you want to install on each Manager.

Installing DataSynapse GridServer

GridServer 7.1.1 is a service pack release. This release ensures compatibility with your existing GridServer 7.1.0 base installation and supports all hotfixes for GridServer 7.1.0 up to and including hotfix 08.

Prerequisites

- You must have GridServer 7.1.0 installed before applying this service pack.
- The 7.1.1 release package is provided as a .zip archive file. Extract the package before installing the service pack.
- If you upgrade from a previous version of GridServer, see Upgrading GridServer. Complete all the steps in the "Upgrading GridServer" section before you begin the installation process in this section.

To install GridServer 7.1.1, perform the following steps:

Procedure

1. Install the GridServer 7.1.0 base. You can apply any 7.1.0 hotfix up to hotfix 08.
2. Apply the GridServer 7.1.1 service pack (`TIB_gridserver_7.1.1.jar`) to the Manager.
3. Start and initialize the Manager.

Task A: Copy Files Before Installation

Before you install TIBCO GridServer®, you must decide where to install the Manager. TIBCO software is installed in a TIBCO_HOME directory, which is referred to in this documentation as TIBCO_HOME and is typically `c:\TIBCO` for Windows, or `/opt/TIBCO` for UNIX. TIBCO GridServer® is installed in an *installation directory* within this, which is referred to as DS_INSTALL and is typically TIBCO_HOME`/datasynapse`. A Manager installation has files in two locations:

- The *Manager directory* is the location of all static, non-volatile files, and is referred to as DS_MANAGER. The recommended Manager directory location is DS_INSTALL`/manager`. Do not install the Manager in a directory with a name containing a space, such as `c:\Program Files`. Also, do not install the Manager in a directory that is a symbolic link to another directory.
- The *data directory* is the location of all volatile files used by the application server, such as server properties and configuration. This is referred to as DS_DATA. By default, these files are installed in a directory created at `DS_INSTALL/manager-data`. To change this location, see Running Managers With a Different Data Directory.

On UNIX systems, the `server.sh` script attempts to create the data directory if it does not exist. If this script fails because it cannot create the data directory, you must either run the script as root, create the data directory with write permissions for the installation user before you run the script, or set the data directory to another location.

Warning: Do not set the data directory to be the same as the installation directory.

To copy files that are used for installing TIBCO GridServer®:

1. The software ships as a gzipped `TAR` archive and a `JAR` file of third-party libraries. The gzipped `TAR` archive is unpacked in the installation directory.
   - On Windows, use WinZip or a similar tool.
   - On UNIX, use `TAR` and run the following command:

     ```
     tar -xvzf TIB_gridserver_7.1.0*.tar.gz
     ```

     Note: The `tar` file contains the `datasynapse` directory. If you want, for example, your installation directory to be `/opt/TIBCO/datasynapse`, extract the archive in `/opt/TIBCO`.
2. Extract the GridServer 7.1.1 release package.

   Note:

   - Ensure that the Manager is not running. For a new installation, the Manager should not be started.
   - Java 11 is required to apply the GridServer 7.1.1.

   Apply the service pack `JAR` for your Manager directory by running the following command:

   ```
   java -jar TIB_gridserver_7.1.1.jar DS_MANAGER
   ```

`where, DS_MANAGER` is the path to the Manager directory. For example, `C:\tibco\datasynapse\manager`.

Service packs are cumulative. You confirm the service pack by verifying the `DS_MANAGER/webapps/livecluster/WEB-INF/config/update.7.1.1.properties` file shows the `7.1.1` build.

3. This is an optional step.

   If you plan on using Windows authentication, after extracting the installation archive, install the third-party library `JAR` with the following command:

   ```
   java -jar TIB_gridserver_version-lgpl.jar path
   ```

   where, *version* is the version of GridServer you are installing, and *path* is the DS_MANAGER path.

4. Install the unlimited strength Java Cryptography Extension (JCE) for your Java SDK. The files are at the following location:

   `DS_MANAGER/webapps/livecluster/WEB-INF/etc/jce`

   Follow the instructions in your SDK's `README.txt`.

Running Managers With a Different Data Directory

All volatile files used by GridServer are stored in the data directory, which is by default `DS_INSTALL/manager-data`. You can change this location by setting the data directory.

To set your data directory:

1. Unpack the installation in the installation directory.
2. Prior to running your Manager for the first time, set the `DS_DATA_DIR` environment variable to the location of the data directory.
3. Run the `server.bat` or `server.sh` script with the `prepare` argument to copy volatile files into the data directory.

Note:

- Before starting the Grid Server, ensure that the Manager is not in a started state. If it is started, then the JSP cache is not deleted. JSP cache is deleted only when the Manager is not in a started state.
- In Windows, update the port number in the `server.bat` file if it is other than the default port 8080.

  The new port number is updated in the following line of the `server.bat` file:

  ```
  netstat -na | find "LISTENING" | find /C /I ":8080" >
  "%DS_DATA_DIR%\tomcatRunningCount.txt
  ```

When specifying a data directory, it cannot contain any spaces in its name, be the same directory as the installation directory, or be a child directory of the `DS_MANAGER/webapps/livecluster` directory. Also, the installation and data directory cannot be on two different Windows drives (for example, `C:` and `T:`).

Task B: Configure Server Settings

If you need to make changes to any of the following, make the changes to the appropriate config file or environment variable, as required:

Java Settings

You must set the `JAVA_HOME` environment variable to the root directory of your Java installation.

Windows

To set the `JAVA_HOME` environment variable:

1. Right click My Computer and select Properties.
2. On the Advanced tab, select Environment Variables, and then add or edit `JAVA_HOME` to point to where the Java JDK installation is located, for example, `C:\Program Files\Java\jdk 11.0.31`.

UNIX

To set the `JAVA_HOME` environment variable, if the Java installation is located in *java-install-dir*:

- Korn and bash shells:

```
export JAVA_HOME=java-install-dir;

export PATH=$JAVA_HOME/bin:$PATH
```

- Bourne shell:

```
JAVA_HOME=java-install-dir;

export JAVA_HOME; PATH=$JAVA_HOME/bin:$PATH; export PATH
```

- C shell:

```
setenv JAVA_HOME java-install-dir;

setenv PATH $JAVA_HOME/bin:$PATH;

export PATH=$JAVA_HOME/bin:$PATH
```

Data Directory

The data directory is the location of volatile files used by the application server, such as server properties and configuration. For more information, see Running Managers With a Different Data Directory.

Port Changes

By default, the Manager uses HTTP on port 8000 for messaging, port 8080 for the web-based Administration Tool, and port 8005 for Tomcat management. You can change any of these locations by editing the `server.xml` file in the DS_DATA`/conf` directory.

For more information, see Changing the HTTP Ports.

HTTPS

By default, all communication takes place over HTTP; you can configure the Manager to use HTTPS instead.

For more information, see Configuring SSL.

Default File Handle Limit

If you are installing a Manager on a UNIX system and plan to run a large grid, you might need to increase the default limit for the number of open file handles. The default on most UNIX systems is 1024, which is insufficient for a busy grid. In Linux, this is done in the `/etc/security/limits.conf` file.

Task C: Start the Manager

Start the Manager according to instructions in the Windows or UNIX section that follows.

Windows

Start the Windows Manager by running the `server.bat` file, located in the DS_MANAGER directory of the GridServer distribution. To test, open a command window and run `server.bat run` to launch the Manager and start GridServer. The command window contains log messages from the application server.

The `server.bat` script takes the following arguments:

ARGUMENT	DESCRIPTION
`start`	Start the Manager. The first time this runs, it copies volatile files into the data directory.
`prepare`	Copy volatile files into the data directory without running the server.

The Manager is typically run as a Windows Service in production systems. To do so, open a command window as administrator and run `service.bat` with the following arguments:

ARGUMENT	DESCRIPTION
`install`	Install the Manager as a Windows service.
`remove`	Remove the Manager’s service.
`start`	Start the installed service.
`stop`	Stop the service.
`--user` *username password*	Can be specified to use a Windows service as another user. For example, to install as user

ARGUMENT	DESCRIPTION
	`jsmith` with password `test123`: `service.bat install --user jsmith test123`

For initial installation, install the Manager as a Windows service, then start the service.

UNIX

Launch the Manager by invoking the `server.sh` script, located in the DS_MANAGER directory of the GridServer distribution, with the following arguments:

ARGUMENT	DESCRIPTION
`run`	Start the Manager in the foreground. The first time this runs, it copies volatile files into the data directory.
`prepare`	Copy volatile files into the data directory without running the server.
`start`	Start the Manager in the background.
`stop`	Stop the Manager running in the background

You can configure the Manager to launch automatically at system startup and stop cleanly at shutdown. Since the `server.sh` script conforms to the standard `start/stop` argument convention (“rc script”), you can accomplish this simply by linking to the appropriate files in the `/etc/rc.d` directories. For more information about initialization and termination scripts, refer to the `init` and `init.d` man pages on your UNIX system and see A Sample UNIX rc.d Script

> **Note:**
>
> When installing the Manager on cloud (AWS), you might have to perform a few additional actions to improve the performance of your system.

> If you are likely to launch a large number of Engines simultaneously, you must apply AWS auto-scaling concepts. Engine installation steps are the same as those described earlier. You can then perform the following steps:

1. Set `JAVA_HOME` to the latest version of JAVA being used by GridServer.
2. Run the following commands to set machine-level properties such as `maxThreads`:

   `sudo ulimit -c unlimited`

   `sudo sysctl -w net.core.somaxconn=1024`

   `sudo sysctl -w net.core.netdev_max_backlog=5000`

   `sudo sysctl -w net.ipv4.tcp_max_syn_backlog=5000`

   `sudo sysctl -w fs.file-max=100000`
3. In the GridServer Administration Tool, navigate to **Broker > Broker config**and change the maximum number of Engines as per your requirement.
4. Go to *DS_DATA* `/conf/` and update the `server.xml` file. Change `maxThreads` and `acceptCount` as per your requirement.

In the following example, we have set the value of `acceptCount` and `maxConnections` as 500:

```
<Connector port="8080" protocol="HTTP/1.1"
```

```
connectionTimeout="20000"
```

```
redirectPort="8443" acceptCount="500" maxConnections="500"/>
```

After the above changes, you must restart the Manager.

The above steps might vary for other cloud service providers such as Azure or GCP.

Task D: Initialize Your Manager

After the Manager is running for the first time, you must initialize it.

To initialize the Manager:

1. Start the GridServer Administration Tool.

   Go to `http://` *yourhost* `:` *port* to open the installation page.

2. Read and accept the End User License Agreement (EULA) to continue installation.

3. Select Installation Type.



   Selecting Installation Type

   Choose a **Standard**or **Custom**Manager installation. The default, **Standard**, installs a primary Director and a Broker with all default settings. If you select **Standard**, click **Next**and skip to step 4 in this procedure.

   Choose **Custom**to install a new Manager (for instance to install a secondary Director, failover Broker, or no Broker) or upgrade an old installation.

4. Enter Configuration Values.

   In the GridServer Manager Configuration page, select **New Manager Installation**, click **Next**, and enter values for each applicable item on each screen. For information about any page, click **Page Help**.

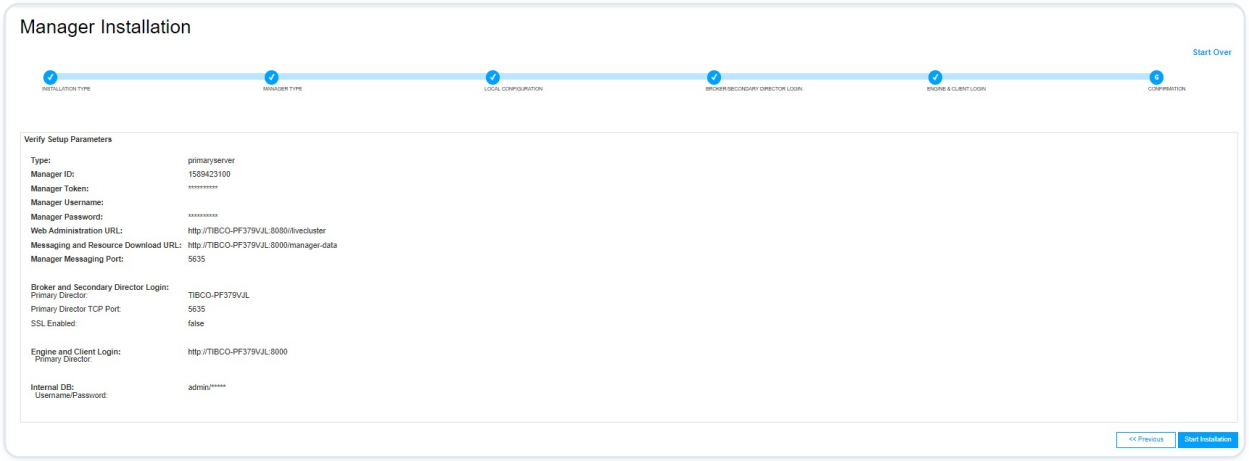
   The following issues might require different configuration than a typical install:

ISSUE	CONFIGURATION INFORMATION
Upgrades	Before installation, move your previous version of GridServer to another location. On the GridServer Manager Setup page, type the path to your old DS_DATA directory in the **Previous Data Directory**box. GridServer migrates your old settings to your new installation. When upgrading from a previous version of GridServer, see Upgrading GridServer and complete all steps required before beginning this installation process.

ISSUE	CONFIGURATION INFORMATION
Installing a Secondary Director or Standalone Broker	Use the two lists on the **Manager type**page to select if you want to create a Manager with no Broker, a Manager with a secondary Director, or a standalone Broker. If you install a secondary Director, reconfigure the Manager containing the primary Director. To do so, go to **Admin > System Admin > Manager Reinstall**and enter the secondary Director's address and port in the corresponding page. This configures the primary Director to recognize the secondary Director and reconfigures the Engine and Driver accordingly.

1. Verify and Install.

   After you complete a **Custom**or **Typical**installation, the **Verify Setup Parameters**screen shows the installation parameters. Review parameters to ensure they are correct. To change parameters, restart the installation process with the **Start Installation**button.



The Manager Installation screen

   When installing the GridServer Manager on a UNIX system, if the installer shows the fully qualified hostname as the Manager name or uses the machine hostname as the full name, this can cause potential routing problems after installation. Change the hostname of the machine with the Custom install option. Go back to the beginning of the installation screens and repeat the installation from Step 2, and enter the correct DNS-resolvable hostname in the **Verify Hostname**screen.

   When satisfied with the parameter values, click **Start Installation**to begin installation.

2. Restart the Manager

   After Manager installation finishes, you are prompted to shut down the application server. On Windows machines, use the `server stop` command; in UNIX, use the `server.sh stop` command. Restart the Manager, using the same command used earlier.

3. Create a root account

   After restarting the application server, go to `http://` *yourhost* `:` *port* again. At the prompts, enter a username, password, and password verification for your initial admin account.

   This first account you create is the *root account*. For more information about the root account, see the TIBCO GridServer® Administration.

   After creating your account, the home page in the Administration Tool appears.

> **Note:**After the server is installed, you can secure `Internal.script` and passwords stored in the `installation.properties` file at `DS_DATA\conf`. For more information, see Securing Passwords and Internal.script.

Silent Installation

The Manager also has a silent installation mode, which can be used to install non-interactively and configure the Manager for automation purposes. In silent mode, the installer does not prompt for any inputs during installation. Instead, the inputs are read from a response file that is provided as a command-line parameter. The silent installer can be used for new Manager installations, or to upgrade or reinstall the Manager.

The `install.silent` file is packaged in the root directory of the TIBCO GridServer® distribution. Edit the file with information for your environment before launching the silent installation. The file includes comments that describe the installation properties you can set. You can also specify properties on the command line that you do not want to put in the properties file (such as passwords).

If errors occur during installation, they are listed in the installation log file at the location DS_MANAGER`/webapps/livecluster/WEB-INF/log/install.log`.

> **Note:**

> You can provide values to the Grid authentication properties during Silent installation for Grid component authentication. For more information, see Authentication of Grid Components.

> To use the silent installer:

1. Perform the following tasks:
   - Task A: Copy Files Before Installation
2. Task B: Configure Server Settings
3. Using a text editor, open the `install.silent` file in the root directory of the TIBCO GridServer® distribution and edit the following properties for the Manager installation.
   - The `acceptEULA` property is set to `false` by default. You must set this to `true` after reading the `EULA.txt` file in the root directory of the TIBCO GridServer® installation.
   - The `passwordEncryption` property is set to `true` by default. This property is used to secure `Internal.script` and passwords stored in the `installation.properties` file at `DS_DATA\conf`. For more information, see Securing Passwords and Internal.script.
   - The `licenseFile` property is blank by default. It means that the license file path is not set. You must set the `licenseFile` property to the full path of your pre-issued (`.ser`) license file. Perform this action before you start the Primary Director. When you configure the license, the license is applied directly into the Manager's internal database. This occurs during silent installation and does not require Director UI interaction. You can update the `licenseFile` property as shown in the following examples:

     For Windows: `licenseFile=C:/licenses/tibco.ser`

     For Linux: `licenseFile=/opt/licenses/tibco.ser`

     > **Note:**If you leave `licenseFile` blank or the property is absent, the installation skips the license step and completes installation. Thereafter, you must manually run the `apply-license` script if you want to update the license post-installation. For more information, refer to "Apply License Using Script" section in the TIBCO GridServer® Upgrade guide.
4. If you are upgrading an existing Manager, ensure that it is not running.
5. Run the `install.sh` or `install.bat` file:

`install.sh/bat` *properties-file* `[` *var1=x var2=y ...* `]`

The *var1...varx* are optional, and enable you to specify properties on the command line instead of in the properties file (such as passwords).

Note:

For upgrades performed with the silent installer, database migration does not overwrite existing entries. If you have configured the Manager and added anything to the database, the upgrade does not overwrite the new data from the previous Manager.

Configuration Options

The following configuration options can be made to customize your Manager installation.

Changing the HTTP Ports

By default, component messaging uses HTTP port 8000, and the web-based Administration Tool uses port 8080. The Tomcat server is also configured to use port 8005 for administration/shutdown.

To change any of the HTTP ports:

1. Before installation, go to the DS_MANAGER directory and run `server prepare` (for Windows) or `server.sh prepare` (for UNIX). This copies all of the application server's volatile files to the data directory.
2. In the `DS_DATA/conf` directory, edit the `server.xml` file.
3. To change the shutdown port, replace the value of 8005 in the following line:

```
<Server port="8005" shutdown="SHUTDOWN">
```

4. To change the port used by the Administration Tool, replace the value of 8080 in the following line:

```
<Connector port="8080"
```

5. To change the messaging port, replace the value of 8000 in the following line:

```
<Connector port="8000"
```

6. Start the Manager and continue installation.

You can verify your changes by opening the new Web Administration URL on the new port in your browser. For the example: http://my-manager:9000

A Sample UNIX rc.d Script

The following is an example of a simple startup script for the GridServer Manager running on a RedHat Linux system:

```
#!/bin/sh

# Startup script for DataSynapse Manager

#

# Source function library.

. /etc/rc.d/init.d/functions

prog="server"

INSTALLDIR=/opt/TIBCO/DataSynapse/manager

DS_DATA_DIR=/var/TIBCO/DataSynapse

export DS_DATA_DIR

JAVA_HOME=/usr/local/java

export JAVA_HOME

case "$1" in

    start)

        cd $INSTALLDIR

        ./server.sh start

        ;;

    stop)
```

```
        cd $INSTALLDIR

        ./server.sh stop

        ;;

    restart)

        cd $INSTALLDIR

        ./server.sh stop

        ./server.sh start

        ;;

    *)

        echo $"Usage: $0 {start|stop|restart}"

        exit 1

esac

exit 0
```

After creating the above file, place it in `/etc/rc.d/init.d/.` Your Linux system does not directly run scripts from this directory. Instead, a different directory within `/etc/rc.d` corresponds to each runlevel of your system. When your system enters a runlevel (for example, during system startup), each script in that runlevel's associated directory runs and receives the `start` or `stop` parameter.

Instead of creating several identical copies of your script, you can create symbolic links in each runlevel directory. Links beginning with `K` run the script with the `stop` parameter; those with `S` run the `start` parameter. The number at the start of each link dictates the order in which scripts run.

The following is an example of the links created when installing the above script. These links start the GridServer Manager at runlevels 3 and 5 and stop it at runlevels 0, 1, and 6:

```
lrwxrwxrwx    1 root     root           18 Apr  8 16:26 /etc/rc.d/rc0.d/
K02datasynapse -> ../init.d/dsserver

lrwxrwxrwx    1 root     root           18 Apr  8 16:27 /etc/rc.d/rc1.d/
K02datasynapse -> ../init.d/dsserver

lrwxrwxrwx    1 root     root           18 Apr  8 16:27 /etc/rc.d/rc3.d/
S98datasynapse -> ../init.d/dsserver

lrwxrwxrwx    1 root     root           18 Apr  8 16:27 /etc/rc.d/rc5.d/
S98datasynapse -> ../init.d/dsserver

lrwxrwxrwx    1 root     root           18 Apr  8 16:27 /etc/rc.d/rc6.d/
K02datasynapse -> ../init.d/dsserver
```

A Sample Systemd Unit File

Starting in Red Hat Enterprise Linux 7, the systemd init daemon is used, instead of the upstart init daemon. This uses the concept of targets defined in a unit file.

The following is an example of a `gridserver.service` unit file for the GridServer Manager running on a RedHat 7 Linux system:

```
[Unit]

Description=Datasynapse Manager service

After=network.service NetworkManager.service NetworkManager-wait-
online.service



[Service]

Type=forking

User=root

PIDFile=/opt/datasynapse/manager-data/java.pid
```

```
Environment=DS_DATA_DIR=/opt/datasynapse/manager-data

Environment=JAVA_HOME=/usr/local/java

WorkingDirectory=/opt/datasynapse/manager

ExecStart=/opt/datasynapse/manager/server.sh start

ExecStop=/opt/datasynapse/manager/server.sh stop



[Install]

WantedBy=default.target
```

After creating the above file and naming it `gridserver.service`, place it in `/etc/systemd/system/`. Refer to the Red Hat documentation for a full understanding of your options for the placement of this file.

The above example sets `WantedBy=default.target,` which creates a "want" dependency for the `default.target` related to this service. If you are using a different target in your environment, you might want to use that target instead of the `default.target` above.

Authentication of Grid Components

Authenticating all Grid components is an additional security option provided in GridServer. To authenticate all Grid components, you must set the following properties in the Manager:

- `Authenticate Manager`
- `Manager Token`
- `Manager UserName`
- `Manager Password`

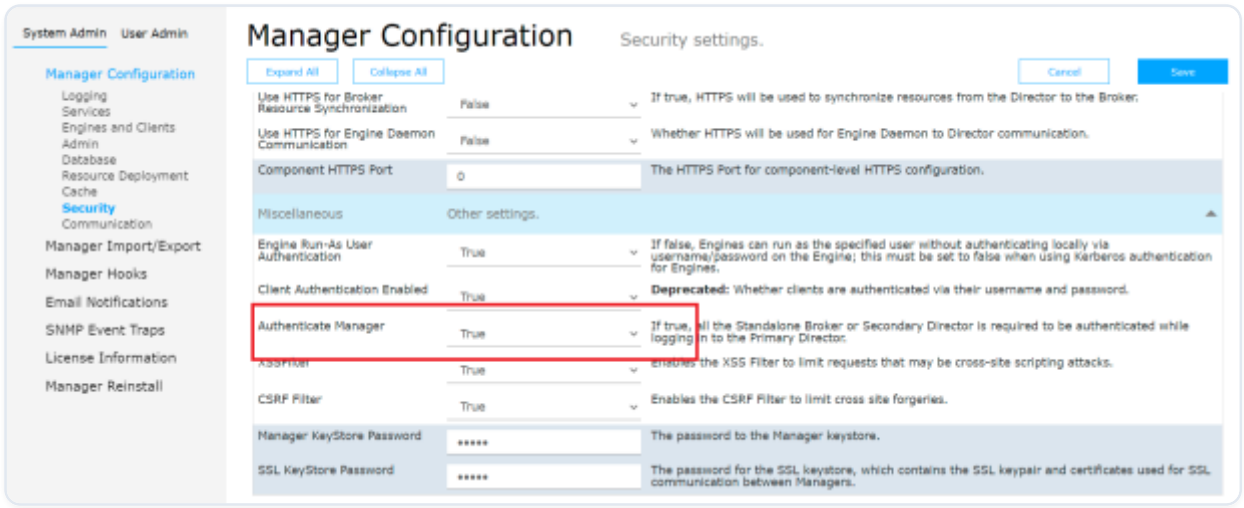
> **Note:**

> You can also provide Grid authentication property values during silent installation for Grid component authentication.

To change the `Authenticate Manager` property, go to the **Admin > System Admin > Security > Miscellaneous**section and set the `Authenticate Manager` property to `False`.

PROPERTY	DESCRIPTION
`Authenticate Manager`	It is set to `True` by default. When logging in to the Primary Director, you must authenticate all Brokers or Primary Directors. If required, you can set this property on the Primary Director or on the Secondary Director.

FIGURE: AUTHENTICATE MANGER PROPERTY



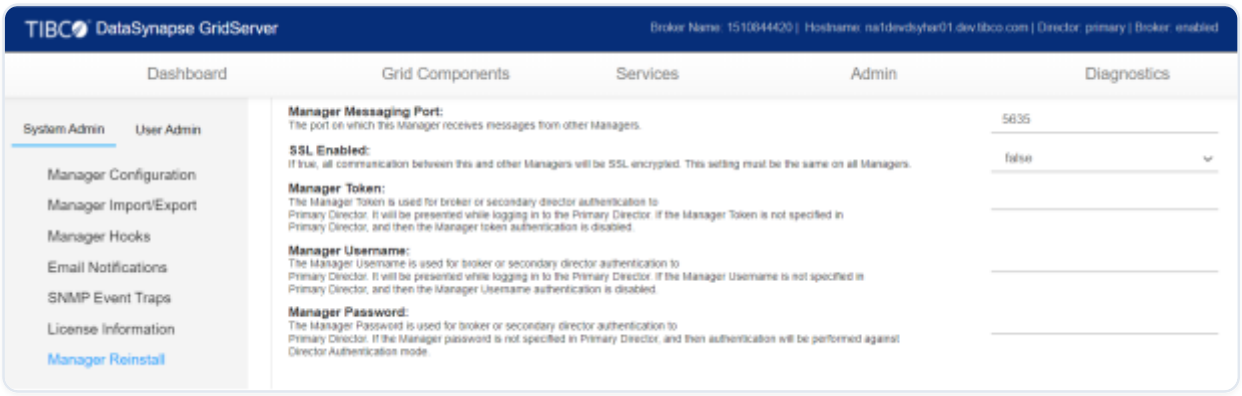
Authenticate Manager property

To configure authentication, set the following properties during installation on the Installation page:

PROPERTY	DESCRIPTION
`Manager Token`	It is used to authenticate the Broker or the Secondary Director when logging in to the Primary Director. If it is not specified in the Primary Director, then this token is disabled.
`Manager UserName`	It is used to authenticate the Broker or the Secondary Director when logging in to the Primary Director. If it is not specified in the Primary Director, then the Manager Username is disabled.

PROPERTY	DESCRIPTION
`Manager Password`	It is used to authenticate Broker or Secondary Director when logging in to the Primary Director. If it is not specified in the Primary Director, then authentication is performed against Director Authentication mode.

FIGURE: GRID COMPONENTS AUTHENTICATION



Configuring Manager Authentication

Points to note

- If you are upgrading the server and using mixed grid deployment then on the Primary Director and Secondary Director, you must set the `Authenticate Manager` property to `false`.
- Grid component authentication fails on the Secondary Director when the Primary Director goes down and the user updates the tokens on the Secondary Director (different from the Primary Director token's values) without a restart.
- The Grid components fail to log in under the following circumstance:
  - If the `Authenticate Manager` property is `true` on the Primary Director and if the token, user name, and password are not configured on the components in sync with the Primary Director.

    In this scenario, the following error is displayed on the Grid Component UI:

    `"Login is not available at this time, the Primary Director may be offline"`.

    In this case, if you do not want to use this feature, then

a. On the Primary Director and the Secondary Director, set the `Authenticate Manager` property to `false` and restart the Primary and Secondary Director (if it exists).

b. Restart the remaining Grid components.

Windows Engine Installation

In the Windows environment, you can download an Engine and install it or transfer it to another computer before installation. You can also use a network installation to install Engines on many machines, or to install multiple Engine instances on different computers , or to install GridServer on a Windows Terminal Server for use by multiple PCs.

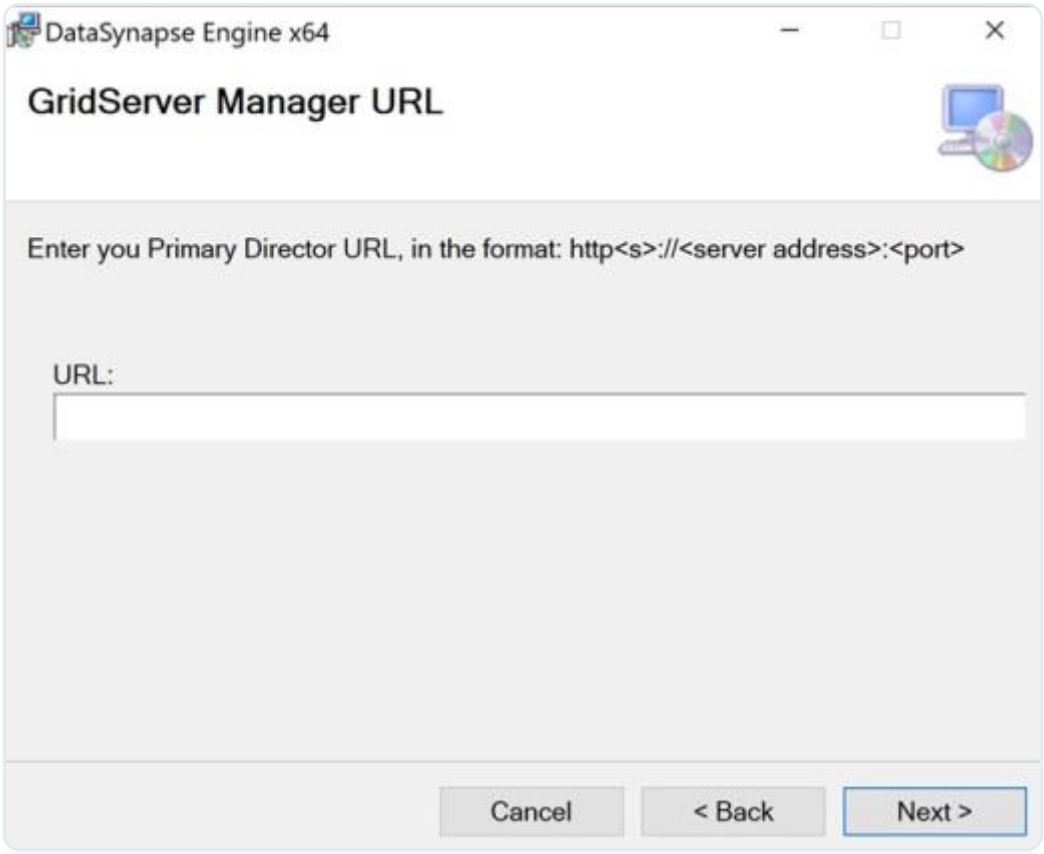
Installing Windows Engines

To install GridServer Engine on a Windows system:

1. Log in to the GridServer Administration Tool, go to the top navigation bar, and click the **Downloads**icon.
2. Click the link to download the 64-bit (x64) Engine archive. This downloads a file named `DSEngineInstall64.msi`.
3. If you are using HTTPS, you might need to copy `ssl.pem` and `ssl.keystore` to the target installation directory, see Configuring SSL.
4. Run the msi installer file, or save it and move it to another machine.
5. Click **Next**on the installer screen.
6. Select the location for the installation and then click **Next**.

   Default installation location is `c:\TIBCO\DataSynapse` by default.
7. Specify the URL of the Manager containing the Primary Director and then click **Next**.

   This must begin with http:// or https:// and include the port number (8000 by default.).



GridServer Manager URL

8. Enter the username and password that must be used to run the Engine's Windows service. Leave the fields blank to use the Local System account. Click **Next**.

> **Note:**Using the Local System account to run the Engine service is discouraged.

9. If you want to enable Engine Daemon authentication during installation, see Enabling Engine Daemon Authentication.
10. To confirm the installation, click **Next**and then click **Close**to exit the installer.

Command Line Unattended Install

To install a Windows Engine from the command line:

1. Copy the `DSEngineInstall64.msi` file in `DS_MANAGER/webapps/livecluster/public_html/register/install/worker/exe` to the Engine machine or a shared directory.
2. If you are using HTTPS, you might need to copy `ssl.pem` and `ssl.keystore` to the target installation directory, see Configuring SSL.
3. If you want to enable Engine Daemon authentication during installation, see Enabling Engine Daemon Authentication.

Here is a sample command:

```
msiexec /quiet /i DSEngineInstall.msi URL=Managerhostname:port
ENGSVCUSER=service username ENGSVCPWD=service password [SSL_PEM=ssl.pem
file path] [LOGINUSER=user] [LOGINPWD=pwd] [DAEMCONFIG=config]
```

Note: Omit `ENGSVCUSER` and `ENGSVCPWD` if the Engine service should run as a Local System.

You can use the following parameters for Windows Engine installation:

PARAMETER	DESCRIPTION
`URL`	Primary Director URL
`ENGSVCUSER`	Name of the user running the Service
`ENGSVCPWD`	Password of the user running the Service
`SSL_PEM`	Optional. Path to directory that contains `ssl.pem`
`DAEMCONFIG`	Optional. Engine Daemon Configuration

Note:

A custom certificate for HTTPS works only with the network installation method. If a Manager uses a custom certificate, you cannot install Engines by interactive installation.

Network Install

In most cases, executing the Windows Engine installation MSI across a network from a shared location works with no issues. If there are problems due to network speed, the contents of the MSI can be expanded into a network share and the installer runs from the network location. The MSI is expanded into a network share by performing an administrative install by adding the `/a` switch to the above `msiexec` command.

During the interactive portion of the administrative install, provide a directory into which the unpacked MSI is stored (the network share from which users install). Users then run the new `DSEngineInstall.msi` from that location (which is smaller than the original) either interactively, or by using the unattended command line parameters.

Engine JRE Synchronization

The Engine's JRE is always kept in synchronization, by default, with its Primary Director. For example, when the Engine is installed, it downloads a JRE from the Director with which it was configured. If the Engine is moved to another grid that has a different JRE, it synchronizes prior to login to that grid.

There might be certain cases in which you might not want the JRE to automatically synchronize. To disable this, add the line `jre=local` to the `intranet.dat` file. This prevents the installer from downloading and unpacking the JRE.

> **Note:**
>
> If JRE updating is disabled, it is the responsibility of the user to make sure the JRE is properly deployed on the Engines.

> If an Engine's configuration uses a different Director URL than the actual name of the Director (such as one using the fully-qualified domain name and the other using a short name), the JRE is not updated when an Engine migrates. Also, if a primary Director is not available and an Engine uses a secondary Directory, the JRE is not updated.

Engine Permissions

Engines installed by an administrator but run by a non-admin user can have permission issues, especially if the Engine removes a registry key and creates it on an update event. This section contains a list of keys created and used by Engines. Use this list to help determine what permissions to set so that Engines function properly.

The following registry keys are used for performance monitoring and require only read access after install:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NetFramework

HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0
```

```
HKEY_DYN_DATA\PerfStats\\StartStat

HKEY_DYN_DATA\PerfStats\\StatData

HKEY_DYN_DATA\PerfStats\StopStat

HKEY_LOCAL_MACHINE\SOFTWARE\microsoft\windows nt\currentversion\perflib
```

Configuring Permissions for Processor Utilization Mode

If you install an Engine Daemon to run as a Service and the Service logs in as another user, the Processor Utilization mode does not work correctly, because the Engine Daemon cannot retrieve performance data (such as CPU information) from the OS.

To remedy this:

1. Start `regedt32` (Click **Run...**from the Start menu, and type `regedt32`.)
2. Navigate to the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Perflib`
3. Right-click the folder named `Perflib` in the left-hand pane and click **Permissions...**
4. Add the user that the Engine logs in as, and grant Read access. Do this for each Engine in the Grid.

Configuring Run-As for Windows Engines

To configure run-as for Windows Engines:

1. Right-click the Engine's install directory, select **Properties**, and under the **Security**tab use **Add...**to add all users that you intend to run Services as.
2. Select the **Allow**checkbox for **Full Control**.
3. In Windows, select **Start > Control Panel > Administrative Tools > Services**. Right-click the Service running the Engine and select **Properties**. You must ensure that the Engine Daemon user can interact with the desktop. If the **Local System**user is selected, select the **Allow Service to Interact with the Desktop**checkbox.
4. Right-click the Engine's `%temp%` directory, select **Properties**, and under the **Security**tab, use **Add...**to add all users that you intend to run Services as.

5. Select the **Allow**checkbox, for **Read**, **Write**, and **Delete**permissions. Note that the **Delete**permission is set using the **Advanced**button on the Security page of the Windows Explorer folder properties dialog.
6. Specify Run-As credentials that have proper security privileges to run the corresponding service type for Engine instances: **User > User Admin > RunAs Credentials**and then add username and password.
7. Add Run-As user for corresponding Service Type on the UI:**Services > Service Types > Edit Service Type for corresponding Service > Add RunAs user**

For more information about using Run-As, see the *TIBCO GridServer® Administration*.

Engine Configuration and Management

For more information about configuring and managing installed Engines, see the *TIBCO GridServer® Administration*.

UNIX Engine Installation

This section describes how to install GridServer Engines for UNIX. To install Engines on UNIX systems, you must download and install an Engine on each system and run a configuration script.

Installing UNIX Engines

Installing the DataSynapse GridServer UNIX Engine comprises unpacking a `gzip`-compressed `tar` archive on a UNIX system; creating or using a profile on the Manager; and associating the Engine with that profile.

To install an Engine on a UNIX system:

1. Log in to the GridServer Administration Tool, go to the top navigation bar, and click the **Downloads**icon.
2. Click the link to download the installation archive appropriate to your hardware and OS.

> **Note:**
>
> A 64-bit UNIX Engine cannot run 32-bit native Services, such as C++ or Java with JNI.

1. Unpack the archive to the directory of your choice. For example, the following extracts the tar file into the directory `/usr/local/DSEngine`:

```
cd /usr/local

gzip -d -c DSEngineLinux.tar.gz | tar xvf –
```

2. If you are using HTTPS, you might need to copy ssl.pem and ssl.keystore to the root directory, see Configuring SSL.
3. Run the `configure.sh` script located in the directory in which you unpacked the archive.

Note:

Running more than one Engine from a shared network mount point is not supported.

The following table lists the `configure.sh` arguments:

configure.sh Arguments

SWITCH	ARGUMENT	DESCRIPTION
`-s`	`Manager:port`	The domain name or IP address for the Primary Director, and HTTP(S) port.
`-P`	`profiles_dir`	Optional target directory for native logging and configuration. The default value is `./profiles`
`-l`	`y|n`	Whether the Primary Director port uses HTTPS.

If you want to enable Engine Daemon authentication during installation, see the Enabling Engine Daemon Authentication topic.

Engine JRE Synchronization

The Engine's JRE is always kept in synchronization, by default, with its Primary Director. For example, when the Engine is installed, it downloads a JRE from the Director with which it was

configured. If the Engine is moved to another grid that has a different JRE, it synchronizes prior to login to that grid.

There might be certain cases in which you might not want the JRE to automatically synchronize. To disable this, add the line `jre=local` to the `intranet.dat` file. This prevents the installer from downloading and unpacking the JRE.

> **Note:**

> If JRE updating is disabled, it is the responsibility of the user to make sure the JRE is properly deployed on the Engines.

> If an Engine's configuration uses a different Director URL than the actual name of the Director (such as one using the fully-qualified domain name and the other using a short name), the JRE is not updated when an Engine migrates. Also, if a primary Director is not available and an Engine uses a secondary Directory, the JRE is not updated.

Configuring Run-As for UNIX Engines

GridServer uses engine PAM authentication service instead of system PAM authentication. Refer to your UNIX-specific PAM configuration instructions to ensure that the Run-As user is properly authenticated.

To set up the dsengine PAM service for Linux, copy the auth and account sections from the login service, dropping the secure tty module. For example:

```
login

#%PAM-1.0

auth required pam_securetty.so

auth include common-auth

auth required pam_nologin.so

auth required pam_mail.so

account include common-account
```

```
password include common-password

session include common-session

session required pam_resmgr.so

dsengine

#%PAM-1.0

auth include common-auth

auth required pam_nologin.so

account include common-account
```

For additional information, see the utilities/testpam2 section of the DataSynapse Customer Support website.

1. Start the Engine:

   ```
   ./engine.sh start
   ```

2. For Linux and Linux64, change mode of all files to be group read/writable:

   ```
   find . | xargs chmod g+u
   ```

3. Change ownership of the `invokeRA` program to `root`, and change it to be set UID:

   ```
   sudo chown root bin/invokeRA

   sudo chmod +s bin/invokeRA
   ```

4. Set the Engine user's umask to make these permissions the default:

   ```
   umask 002
   ```

Note that for UNIX engines, the Run-As user's account must be in the same group as the Engine user's account.

For more information about using Run-As, see the *TIBCO GridServer® Administration*.

Running the Engine

The `engine.sh` script follows the standard start/stop argument convention for system initialization scripts (“rc scripts”), so you can incorporate it in the start-up and shutdown sequence by inserting appropriate links in the `/etc/rc.d` files.

The `engine.sh` arguments are as follows:

engine.sh Arguments

SWITCH	DESCRIPTION
`start`	Starts the Engine
`stop`	Stops the Engine
`startfg`	Starts the Engine, and runs it in the foreground. Useful for monitoring or debugging purposes.

By default, Engine uses default configuration. To change the default configuration, use `config` argument when starting the Engine:

```
./engine.sh start -config=configValue
```

or

```
./engines.sh startfg -config=configValue
```

Note:

If you install a machine without a default route, the UNIX Engine returns 0.0.0.0 as the IP address. An IP address of 0.0.0.0 causes communication issues with the Manager. You must ensure that a default route is set for Engines to operate properly.

Engine Configuration and Management

For more information about configuring and managing Engines, see the *TIBCO GridServer® Administration Guide*.

Enabling Engine Daemon Authentication

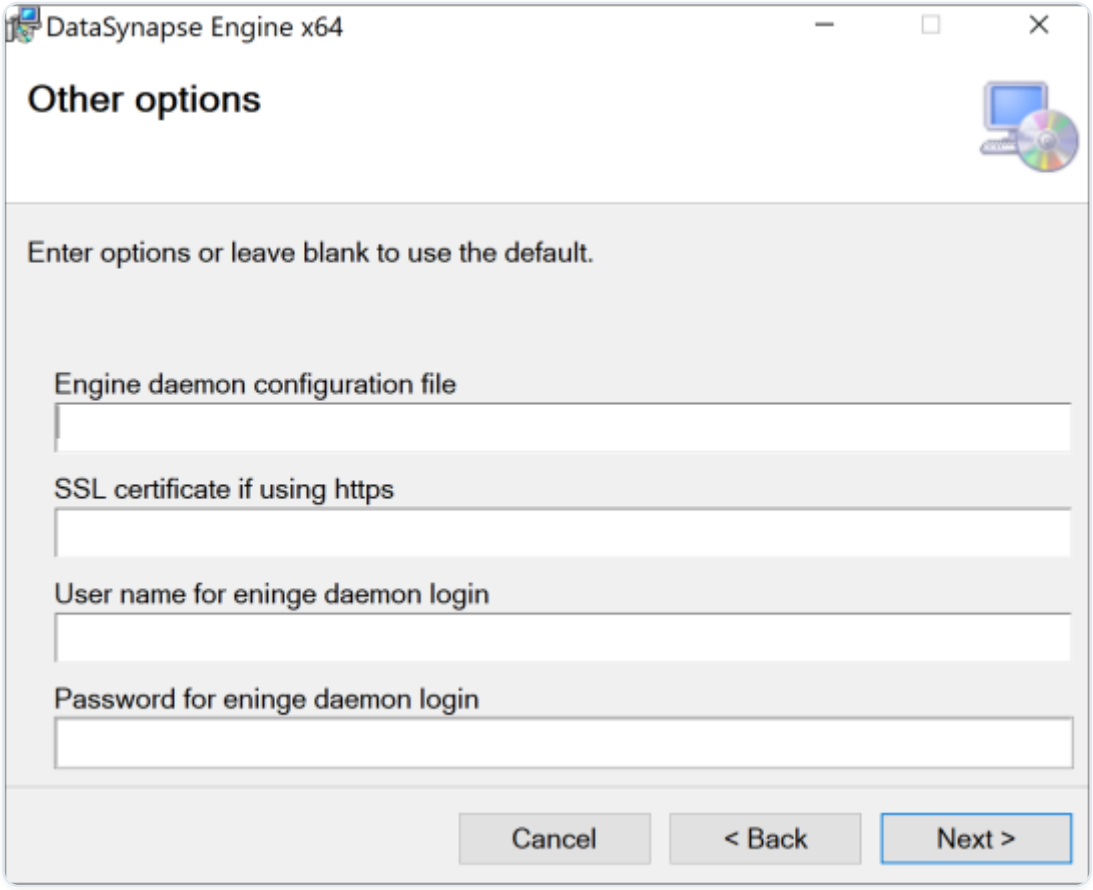
You can enable Engine Daemon authentication during Engine installation. This feature is supported only for Authentication Mode, (**Admin > User Admin > Authentication**), configured as Internal DB or LDAP.

1. Go to **Admin > System Admin > Manager Configuration > Engine and Clients**and set **Daemon Authenticate**to true.
2. Create a user with a Security Role with the **Install Engine Daemon**feature enabled. Ensure that the user has a password that is at least eight characters long.
3. During the Engine installation, depending on your operating system, perform one of the following procedures:

> **Note:**If the Engine is installed with an invalid username or password, the Engine Daemon stops.

> **For Windows by using the Installer**

Enter the configuration, username, and password in the **Other options**dialog.



Engine Daemon Authentication

For more information about installing the Windows Engine by using the Windows Engine Installer, see Installing Windows Engines.

For Windows by using the Command Line Interface

To enable Engine Daemon Authentication by using the command line, run the following command:

```
msiexec /quiet /i DSEngineInstall.msi URL=Managerhostname:port

LOGINUSER=user LOGINPWD=pwd DAEMCONFIG=config
```

PARAMETER	DESCRIPTION
`LOGINUSER`	Username for Engine Daemon login
`LOGINPWD`	Password for Engine Daemon login

For more information about installing the Windows Engine by using the Windows command line, see Command Line Unattended Install.

For UNIX

Use the following command:

```
./engine.sh start -installuser=user -installpwd=password -config=myConfig\
```

The following table lists the `engine.sh` arguments:

engine.sh Arguments

ARGUMENT	DESCRIPTION
`installuser`	Username for Engine Daemon login
`installpwd`	Password for Engine Daemon login

For more information about installing Unix Engine, see Installing UNIX Engines .

To Reset the Username or Password

1. Delete the `DefaultInstallOption.dat` file present in the `Engine` directory.
2. From the command line, start the Engine based on the operating system:
   - For Windows, use the following command: `Engine.exe start -installuser=user -installpwd=password -config=myConfig`
   - For UNIX, use the following command: `./engine.sh start -installuser=user -installpwd=password -config=myConfig\`

   Alternatively, you can also perform the following steps to reset the username or password:
3. Open the `DefaultInstallOption.dat` file present in the `Engine` directory.
4. Delete all the content and then add the username or password as follows: `installuser=user\ installpwd= pwd`

Verify Installation

After installing a GridServer Manager and one or more Engines, you can test your installation by submitting a test Service and watching its progress in the GridServer Administration Tool.

Submitting a Service Test

A Service Test can run a Linpack benchmark that you can use as a basic test for your Grid.

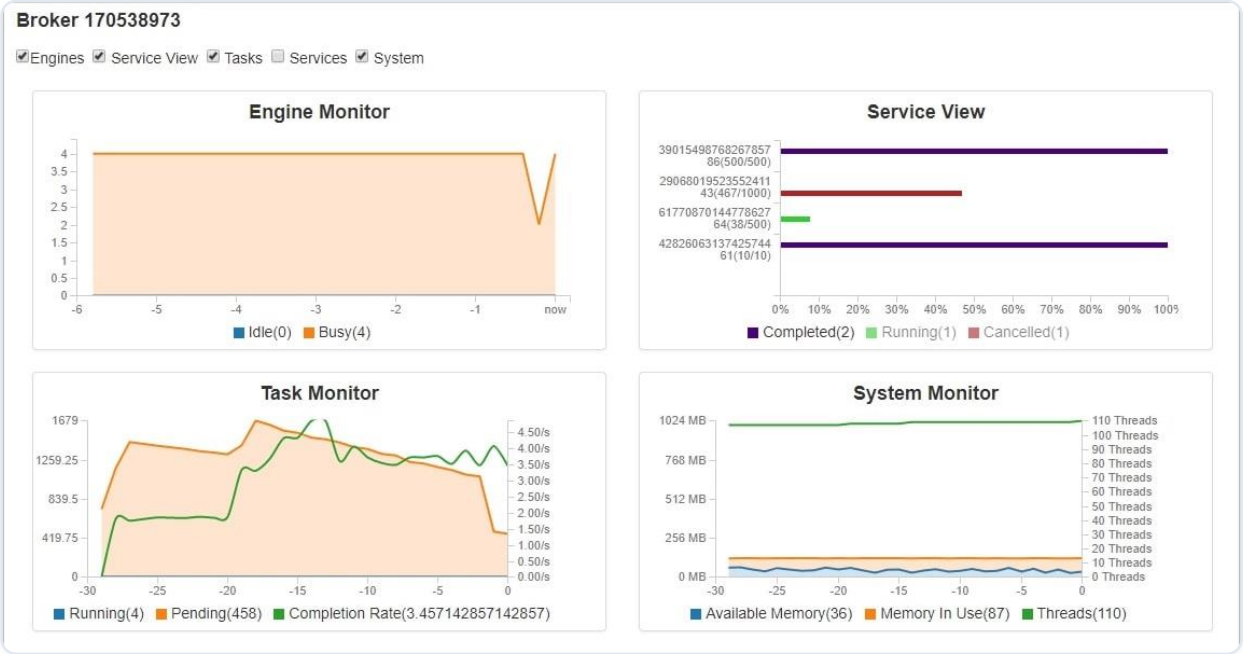
To submit a Service Test:

1. Make sure at least one Engine is logged in to your Manager.
2. Log in to the GridServer Administration Tool and go to **Services > Services > Service Test**.
3. Click **Submit**to send the Linpack Test with default values. You can also change the number of Tasks or Duration to make a Service that takes longer to complete.
4. The **Services > Services > Service Session Admin**page appears. From this page, you can monitor the Service's progress.

Monitoring a Service Test

After you submit the Service Test, you can monitor its progress to ensure that your Engine is taking tasks and returning results. While the Service Test is running, you can also view other pages for more information:

- Go to **Grid Components > Brokers > Broker Admin**. A list of available Brokers appears, along with the number of busy, idle, and total Engines reporting to that Broker.
- Go to **Dashboard > Broker Monitor**. A screen appears showing four interactive graphs: Engines, Service View, Tasks, and System. The System Monitor graph shows memory and thread activity.



The Broker Monitor

- Go to **Grid Components > Engines > Engine Admin**for a list of all Engines and their status.

Troubleshooting

If the Service Test does not function, or you have other issues relating to your installation, consult the TIBCO GridServer® Administration and online help in the Administration Tool.

Uninstalling GridServer

This section describes how to uninstall GridServer. The uninstall procedures for the Windows operating system and the UNIX operating system differ.

Uninstalling GridServer on Windows Systems

Uninstalling an Engine

To uninstall an Engine on a Windows system, first ensure that the Engine is stopped and then go to **Start > Control Panel > Add/Remove Programs or Start > Control Panel > Programs and**

Features. Remove the **DataSynapse Engine**. Follow the onscreen instructions through the uninstall.

Uninstalling GridServer on Windows Systems

Uninstalling an Engine

To uninstall an Engine on a Windows system, first ensure that the Engine is stopped and then go to **Start > Control Panel > Add/Remove Programs or Start > Control Panel > Programs and Features**. Remove the **DataSynapse Engine**. Follow the onscreen instructions through the uninstall.

Manual Uninstall

Manual Uninstall

Uninstall can be done from the command line as follows:

```
msiexec /x DSEngineInstall.msi
```

or

```
msiexec /quiet /x DSEngineInstall.msi
```

Instead of passing the name of the MSI file to `msiexec`, you can alternately substitute the Product Code, which is `{F80D0E61-23AF-4934-A40B-190F2A1C52CE}` for 64-bit Engine. For example:

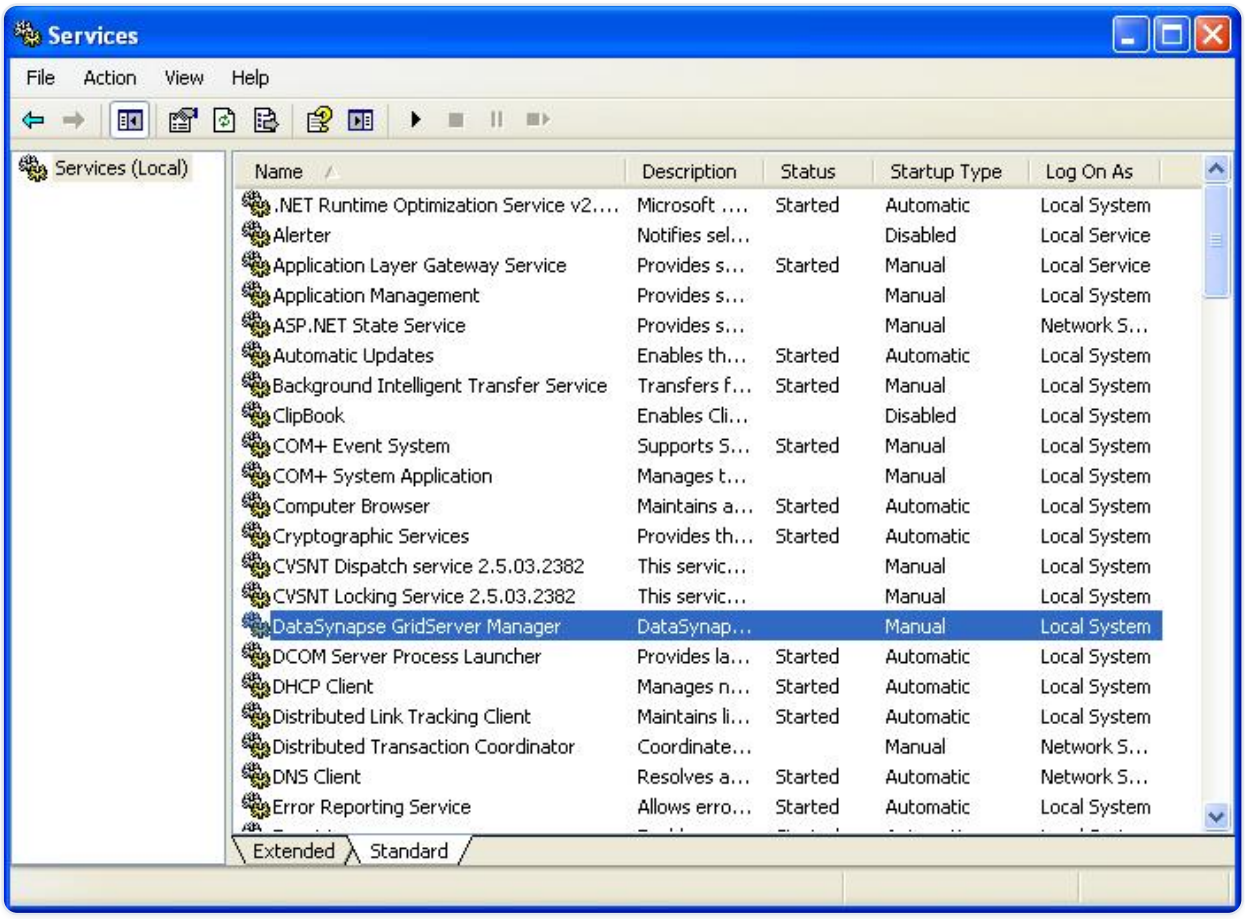
```
msiexec /x {F80D0E61-23AF-4934-A40B-190F2A1C52CE}
```

Uninstalling a Manager

Uninstalling a Manager

To uninstall a Manager installed as a service on a Windows system:

1. In Windows, go to **Start > Control Panel > Administrative Tools > Services**.
2. Click the **Standard**tab.
3. Click **DataSynapse Manager**and click the Stop icon to stop the application.



Stopping a Manager installed as a Windows Service

4. Open a command line, go to the DS_MANAGER directory, and issue a `service remove` command to uninstall the service.
5. Remove the DS_INSTALL directory and its contents.

Uninstalling GridServer on UNIX Systems

To remove a GridServer Manager or Engines on a UNIX system:

1. Stop the Engine or the Broker (`engine.sh stop` or `server.sh stop`).
2. Remove any modifications you made to startup files, typically located in `/etc/rc.d` directories.
3. Remove the DS_INSTALL directory and its contents.

Uninstalling a Service Pack

When you apply a service pack, the system backs up every file it replaces. For example, the system saves the files replaced by service pack `7.1.1.xx` in the `DS_MANAGER/updates/7.1.1.xx/manager` directory. The `UPDATE_DIR/update-summary.txt` file records all files added or removed by the hotfix.

To uninstall a service pack, perform the following steps:

Procedure

1. Shut down the GridServer Manager.
2. Restore all files from `UPDATE_DIR/manager` back to `DS_MANAGER`.
3. In `update-summary.txt`, search for the list of files that the hotfix added. Delete those files from `DS_MANAGER`.
4. Reverse any manual steps that the hotfix README instructed you to perform.
5. Start the GridServer Manager.

Upgrading GridServer

This section explains how to upgrade GridServer components. It contains a GridServer Upgrade Checklist, which explains what to do before, during, and after your new GridServer installation to ensure carrying over old values to the new installation, when possible.

Requirements Changes

See the system requirements in the TIBCO GridServer® readme file to ensure that your software and hardware meet the minimum system requirements. Ensure to review the full list of requirements.

GridServer 7.1.1 Upgrade Checklist

You can migrate settings from your previous version of GridServer to your new version of a GridServer installation, but it's important to not install on top of your previous installation. Instead, shut down your old Manager, and rename the old installation directory. You can then install the new version of GridServer in the same location. Later in the install, you supply a path to the renamed old installation so settings can be copied. An `upgrade.log` file is generated that lists all changed configuration values, as well as settings that could not be migrated to the new installation.

It is recommended that before you start the upgrade, back up the `server.xml` file from the `DS_DATA/conf` directory to a separate directory to preserve your custom configurations. After the upgrade, replace the `server.xml` file in `DS_DATA/conf` with the updated file from the release. Then, reapply any required customizations.

The only direct upgrade supported on GridServer version 7.1.1 is from GridServer versions 7.1.0. To upgrade from any other earlier version to GridServer 7.1.1, you must first upgrade to GridServer version 7.1.0.

Note:

- .

  Authenticating all Grid Components is an additional security option provided in GridServer. For more information, see Authentication of Grid Components.
- Migrate any deployed web components, Broker or Manager hooks, or bundled libraries that reference the `javax.*` servlet API to the `jakarta.*` namespace.

Prior to Upgrade

- If you have any UNIX Engines that share a common installation directory, you must migrate those installations to a local directory. A shared install point is no longer supported.
- To enable programmatic version control, ensure that `gridLibraryStrictVersioning` is set to `false`.
- If you have a Linux64 Engine, for upgrade from all versions earlier than 7.0.0_hotfix08, you must perform the following steps:

1. Stop the server.
2. As a best practice, take a backup of the existing `engine.sh` file.
3. Replace the `engine.sh` file from the `engineScriptUpdate` folder to `datasynapse/manager/webapps/livecluster/engineUpdate/linux64/ and datasynapse/manager-data/engineUpdate/linux64/` folder.
4. Start the server.
5. Let the Engines log in and then restart the Engine Daemon.
6. Stop the server.
7. Upgrade the server.

Note: In case of Grid Architecture, perform the steps from step a to step g on the Primary Director.

- Check the list of changes in the release notes for any other issues that might affect you.

After Stopping the Manager

- When running GridServer as a Windows service, remove the service prior to upgrading.

  The following steps are necessary only if you set up the system by using an external database:

- Copy the corresponding database properties file from old `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db/` directory to the current installation. This is used when you run the database upgrade script.
- Upgrade your external reporting database schema using the `upgradedb.sh` or `upgradedb.bat` script located in `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db`.

During Installation

- For **Installation Type**, choose **Custom, Manager Upgrade**.
- For **Previous Data Directory**, enter the pathname of your previous base directory.

After Installation

If Using System Classloader for Engines

If you are using the system classloader in your Engine configuration and deployed the `DSEngine.jar` file to Engines, deploy the new `DSEngine.jar` to Engines after upgrade. This is now deployed in `DS_MANAGER/webapps/livecluster/engineUpdate/shared.` If `DSEngine.jar` is deployed in the default resources directory, it has already been removed. If it is deployed to any Grid Libraries, you must remove it from them.

External Database JDBC Drivers

If you are using an external database, copy the required JDBC driver JAR archive from `DS_MANAGER/webapps/livecluster/WEB-INF/lib` in the old installation to `DS_MANAGER/webapps/livecluster/WEB-INF/lib/` in the new installation.

Apply License Using Script

For an existing installation, run the `apply-license` script after the installation completes and before you start the Primary Director. The `apply-license` script is at the following location:

Windows: `DS_MANAGER\apply-license\apply-license.bat <license.ser>`

Linux: `DS_MANAGER/apply-license/apply-license.sh <license.ser>`

> **Note:**

- Ensure that the Primary Director is not running when you run this script. If Tomcat is already started, it holds the database file lock, which causes the script to fail.
- You can obtain the license file from DataSynapse Support.

LDAP Upgrade

Starting with GridServer 6.0, Driver profiles are no longer supported. The `allowedBrokers` property has been replaced by the value of **Manager List**in the security role assigned to the Driver user. Also, users are now allowed to run Services based on the "Execute Services" permission rather than by virtue of having a profile.

During upgrade, a new role is created for each Driver profile. This role has execute permission, and maps the "Allowed Brokers" field to "Manager List". That role is added to each user who has the corresponding profile.

You are not required to maintain these converted profiles; you might choose to reorganize and remap these roles as you see fit.

If you use LDAP authentication and you prefer to assign roles via LDAP groups, you must add those groups to your users in LDAP, and remove the locally assigned roles.

Users and roles are logged during the upgrade to the file `DS_DATA/logs/newRoles.txt`, so that you have a list of groups to add the user's LDAP entries. It is a flat file with the following format:

```
[user]\t[role1,role2,...]
```

For example:

```
joe driver_developers,driver_boston

tom driver_testing,driver_london
```

Engine Upgrade

- For UNIX Engines, `bin/invokeRA` cannot be automatically upgraded. After upgrading, you must manually copy the new `bin/invokeRA` to the upgraded Engine, and follow the instructions in "Using Run-As" in the TIBCO GridServer® Administration.

Driver Upgrade

- Download the new version of the GridServer SDK in the GridServer Administration Tool.
- Make sure that all Drivers are upgraded. This simply involves replacing the libraries or executables with the new version.
- If you have existing CPPDriver applications, relink them with the new libraries included in the SDK.
- On the Driver side, link your code with the `dsUtil` library. Link Engine-side Service code with the `dsUtil` library if it uses any exception classes.

> **Note:**
>
> After the server is upgraded, you can secure the `Internal.script` and passwords stored in the `installation.properties` file at `DS_DATA\conf`. For more information, see Securing Passwords and Internal.script in the TIBCO GridServer® Manager Installation Guide.

Mixed-Version Deployments

Mixed-version deployments are grids that consist of GridServer Manager versions 7.1.0. This is an allowed configuration, as a convenience for large Grids that cannot upgrade all Brokers at once. You must address the following issues when running mixed-version deployments:

- The Directors must be upgraded first to the latest version of GridServer Manager 7.1.0.
- All Brokers must be on the latest service pack and hotfix of the minor version.
- You must disable client version checking for Engines, Drivers, and Brokers at **Admin > System Admin > Manager Configuration > Admin**.
- Hard-partitioning of Engines is required. That is, Engines must not be allowed on the GridServer Manager version 7.1.0 Broker. This is done with the balancer configuration.
- You must deploy all versions of the C++ Bridge Grid Libraries via the Primary Director. If you upgraded the Director, the older bridges might already be there, but if the Director is a new installation you must copy them from your old Managers. The Engines automatically choose the proper library based on the Broker version. Because both versions of the C++ Bridges are used, Grid Libraries must also be used on an older Broker if the code is native, rather than using the standard resource deployment.

Mixed-Version Driver Compatibility

GridServer Manager 7.1.1 has backward compatibility with GridServer Manager 7.1.0.x Drivers.

Rolling Upgrades

Use the following procedures to install an update to a large grid (a grid with a large number of Engines and several Brokers) in situations where you cannot permit grid-wide Engine restarts and interruptions in Service availability. These instructions enable Services already deployed on components using a previous version of GridServer Manager to continue running as other components on the grid are updated.

Prerequisite Actions

For all Managers:

1. Bring all Managers up to the required hotfix level.
2. Confirm that all GridServer Managers are at that level. On the Primary Director, you can get the versions of all Brokers on the **Grid Components > Brokers > Broker Admin**page. To find the version of the Directors, click the **About**link at the top of the Administration Tool.
3. In each Engine Configuration, confirm that **Log off Engines on Daemon Logoff**is set to false (the default setting). This prevents unnecessary Engine restarts when the Director is restarted.
4. On each Broker, go to **Admin > System Admin > Manager Configuration > Admin**and set **Daemon Upgrade From Broker**to false; then go to **Admin > System Admin > Manager Configuration > Resource Deployment > Broker Settings**and set **Synchronize Resources From Director**to false. This prevents all Daemons from restarting to upgrade when the Primary Director is upgraded.
5. On each Director, go to **Admin > System Admin > Manager Configuration > Admin**. Confirm that **Daemon Upgrade From Director**is false. (This is the default and would rarely be true.)
6. To assist in the event of any issues, set the logging levels on all Managers and Engines to Fine.

Upgrading a Grid

Note that each Manager upgrade procedure does not make any modifications to the previous installation. If required, you can use this directory to roll back the upgrade.

Primary Director Upgrade

1. Shut down the Primary Director. The Engine Daemons move to the Secondary Director and the Engine Instances (which might be running tasks) continue without interruption.

2. Upgrade the Primary Director.
3. Start the Primary Director.
4. Verify the the following Director and Broker mismatch controls are True: **Admin > System Admin > Manager Configuration > Admin > Version Management >Allow Director Version Mismatch**and **Admin > System Admin > Manager Configuration > Admin > Version Management >Allow Broker Version Mismatch**.
5. Check that all the Brokers are logged in to it at **Grid Components > Brokers > Broker Admin**. The Engine Daemons move to the Primary Director. The Engine Instances do not restart.

Secondary Director Upgrade

Follow the above procedure for the Primary Director. Shut down the Secondary Director and upgrade.

Broker Upgrades

Failover Brokers must be upgraded after regular Brokers.

For each Broker:

1. Shut down the Broker. Within a few minutes Engines and Drivers migrate to a Failover Broker.
2. Upgrade the Broker.
3. Start the Broker. Verify that it logs in to the Directors.
4. Verify that **Admin > System Admin > Manager Configuration > Admin > Version Management > Allow Driver Version Mismatch**is True.
5. Within a few minutes Engines and Drivers migrate back.

Upgrade the Engine Daemons

At this point, the Grid has been updated, but Engine Daemons are still at the previous version. Daemon upgrades result in Engine restarts, so do the following on each Broker when the time is appropriate:

- On each Broker, go to **Admin > System Admin > Manager Configuration > Admin**. Set **Daemon Upgrade From Broker**to true.

> **Note:**The above step needs to be done as soon as possible.

Installing Hotfixes

To install or uninstall a hotfix or service pack, follow the instructions in the readme file distributed with the hotfix.

Securing Passwords and Internal.script

You can enable or disable the encryption of the internal database password and the reporting database password stored in the `installation.properties` file. You can also make the `Internal.script` file available in an unreadable form.

The Grid Server can be installed or upgraded in the GUI installation mode and silent installation mode. The procedure to enable encryption for these installation modes is explained later in this section.

Considerations for Enabling Encryption

- **For the Internal.script file:**This change is only for Primary Director and Secondary Director because there is no separate internal database for Brokers. If you set the property to `true` on the Primary Director and `false` on the Secondary Director, the `Internal.script` on the Secondary Director becomes unreadable. This happens because the Secondary Director backs up the database from the Primary Director.
- *For mixed Grid Deployment:* In case the property is set to true, mixed deployment of the Grid is not supported. Ensure that all components use GridServer Manager version 7.1.1.

Enable or Disable Encryption (GUI Installation Mode)

1. Install or upgrade the server as required.
2. Go to the `DS_DATA\conf` folder and open the `installation.properties` file for editing.
3. In the `installation.properties` file, set the `DSPasswordEncryption` property to `true` (to enable encryption) or `false` (to disable encryption) as required.
4. Reinstall the server by navigating at **Admin > Manager Reinstall**.

> **Note:**You can change the property after the installation. To do so, perform steps 2 to 4.

Enable or Disable Encryption (Silent Installation Mode)

1. Go to the `DS_MANAGER` folder and open the `install.silent` file for editing.

2. In the `install.silent` file, set the `passwordEncryption` property value to `true` (to enable encryption) or `false` (to disable encryption) as required.
3. Install or upgrade the server as required.

Note:

- You can change the property after the installation. To do so, perform steps 1 to 3. Then, provide the `managerId` in the `install.silent` file and reinstall the server.
- The `managerId` is in the `installation.properties` file at `DS_DATA\conf`.
- It is optional to provide the `managerId` when you install the Grid Server for the first time. But, it is mandatory when you reinstall the server.

Configuring SSL

This section provides information about configuring SSL to provide secure communication between GridServer components.

About SSL

SSL (Secure Socket Layer) communication can be enabled for communication at each level in the GridServer architecture depending on the security requirements of the organization and the deployment scenarios involved. SSL provides both encryption of messaging between components, and a trust relationship of the server by the client. SSL over sockets can be used for Manager inter-communications, and HTTPS (HTTP over SSL) can be used for all other components and for resource synchronization.

Communication Overview

To understand how SSL is used for messaging, it is important to understand how components establish communication channels with each other. For the remainder of this discussion, the terms “client” and “server” are used in the traditional way, that is, a client/server relationship. An example is the Engine Daemon is a “client” to the Director’s “server”.

There are two aspects to establishing communication. The first step is the login process. The client requests a login through a known communication channel. At that point, the server might perform authentication or validation, and if successful, it returns a connection for use from then on. Note

that this channel might be on a different server. For example, an Engine logs in to a Director, but the connection exists on a Broker.

SSL is configurable for both aspects. If SSL is to be used for login, it must be configured on the client. If SSL is to be used for the connection, it must be enabled on the server. For example, to enable a Driver to login using SSL, the Driver must be set to the HTTPS URL address on the Director, either in the `driver.properties` file or with the API. To enable HTTPS communication between the Driver and Broker after login, it must be set on the Broker, typically by configuring all Messaging and Download URLs to the HTTPS URL.

Certificate Overview

All SSL clients establish a trust relationship with their server. This is performed with a certificate on the client side, which is a public key that is associated with a private key on the server. When establishing the trust relationship, the server's certificate must either have been signed by a key trusted by the client, or be trusted explicitly by the client (a self-signed certificate). Most SSL clients contain a set of trusted Certificate Authorities (CAs), so that if a server has a certificate signed by one of those CAs, it automatically trusts the server. If the server is self-signed, that server's certificate must be added to the client's list of trusted servers.

In addition, the client checks the Common Name (CN) of the server's certificate against the hostname of the server, to verify that the certificate is being used on the intended host. This means that all Servers require a certificate chained to the top-level certificate.

For Java clients, such as Java Drivers and Engines, the certs are kept in a trust keystore. The JRE is shipped with a default keystore called *cacerts*. To add a certificate, typically this file is copied, the certificate is imported, and the JVM is pointed to the new file.

Native clients that use the cURL library use a platform-dependent directory that contains CA certificates. GridServer Engine Daemons and C++ Drivers also look for a PEM-encoded certificate called `ssl.pem` in the current working directory.

.NET clients look in the Windows Trust Store local to that machine. Certificates can be added to that store by double-clicking on the file and importing using the wizard.

For demonstration purposes, GridServer is packaged with a demo self-signed key-pair and certificate. All clients have a local copy of the certificate added to their list of trusted servers. While this self-signed cert is useful to demonstrate how to set up SSL, a certificate for your organization must be used in a production environment.

Key and Certificate Location and Generation

The following is a list of all key and certificate files used by the Servers and Clients.

Key and Certificate Files

FILE	LOCATIONS	DESCRIPTION
`server.key`	`DS_DATA/certs`	The private key used by OpenSSL on Tomcat for HTTPS.
`server.crt`	`DS_DATA/certs`	The certificate used by OpenSSL on Tomcat for HTTPS.
`ssl.pem`	• `DS_DATA/engineUpdate/shared` • In the current working directory of the C++ Driver	Used by native clients when self-signed.
`server.keystore`	`DS_DATA/certs`	A keystore that contains the `server.key`, used by Manager Socket SSL.
`ssl.keystore`	• `DS_DATA/engineUpdate/shared` • `DS_DATA/certs` • In the working directory or classpath of the Java Driver	A keystore that contains the root certificate and all CA certs, used by Manager Socket SSL, Java Drivers, and Engines, when self-signed.
`manager.keystore`	`DS_DATA/certs`	The keystore used by the Manager for encrypting data, such as user

FILE	LOCATIONS	DESCRIPTION
		passwords. It is not used for SSL.

Manager HTTPS Configuration

The HTTP and HTTPS connections are configured in the Tomcat configuration file, `DS_DATA/conf/server.xml.` By default, there are four connector elements: two HTTP elements and two commented out HTTPS elements. On a Manager Install or Reinstall, this file is read; the first element is used as the Administration Tool connector, and the second is the messaging connection. These connectors are mapped to a Manager as follows:

- Administration Tool Connector: Any time a user attempts to log in over any other connector, the user is redirected to this connector.
- Messaging Connector: The Director URL is set to use this connector, as are the connections established after log in.

Configuring HTTPS

The following procedure is used to configure HTTPS:

Task A Prepare Certificates

1. Create a temporary directory (such as `/opt/mycerts`) and execute the following commands within it.
2. Copy the `cacerts` file from the JDK to the file `ssl.keystore`.

   For JDK8:

   ```
   cp $JAVA_HOME/jre/lib/security/cacerts ssl.keystore
   ```

   For JDK11:

   ```
   cp $JAVA_HOME/lib/security/cacerts ssl.keystore
   ```

3. If you have a signed certificate:
4. If your certificate is from a well-known CA whose root certificate is already in `$JAVA_HOME/jre/lib/security/cacerts`, it is in `ssl.keystore`.

If it is not, add your CA's root certificate to the `ssl.keystore`:

```
keytool -importcert -file CA_ROOT.crt -keystore \

ssl.keystore -alias CA_ROOT
```

5. If you have a signed certificate `.pem` file and the private `.key` file that was used to generate the certificate request, copy or rename them to `server.crt` and `server.key` respectively.
6. If you don’t have a CA signed certificate and need a self-signed certificate, generate a self-signed certificate (and its associated private key):

```
openssl req -x509 -sha256 -nodes -days 1826 -newkey \

  rsa:2048 -keyout server.key -out server.crt
```

7. If you are using a self-signed certificate, import the new certificate into `ssl.keystore`:

```
keytool -importcert -file server.crt -keystore \

ssl.keystore -alias MySelfSigned -storepass changeit
```

8. Copy the `server.crt` file to `ssl.pem`:

```
cp server.crt ssl.pem
```

9. Create a PKCS #12 format file that contains your keypair:

```
openssl pkcs12 -export -in server.crt -inkey server.key -out
server.pkcs12
```

10. Convert it to a JKS format keystore:

```
keytool -importkeystore -srckeystore server.pkcs12 -srcstoretype pkcs12
-srcstorepass changeit -destkeystore server.keystore -deststoretype jks
-deststorepass changeit
```

11. Copy the files into your Manager installation:

```
cp ssl.keystore ssl.pem DS_DATA/engineUpdate/shared

cp server.crt server.key ssl.keystore server.keystore \

   DS_DATA/certs
```

12. If you are enabling SSL on both a standalone Director and standalone Broker (both installed on different hosts), the previous step must be followed for each individual Manager installation. Also, while enabling SSL on a standalone Broker, after following the steps listed above, the standalone Director's `server.crt` must also be imported to the `ssl.keystore` of the standalone Broker:

    ```
    keytool -importcert -file dserver.crt -keystore ssl.keystore \

      -alias MySelfSignedDir -storepass changeit
    ```

    Here, the `dserver.crt` is the `server.crt` from the standalone Director.

13. Copy the latest `ssl.keystore` and `ssl.pem` files on standalone Director and Broker's directory.

    ```
    cat dserver.crt server.crt<Broker server.crt>  > ssl.pem
    ```

Task B Enable HTTPS/SSL on the Manager

1. Edit the `DS_DATA/conf/server.xml` file:
2. Comment out or remove the two connector elements for ports 8000 and 8080.
3. Uncomment the two connectors for ports 8443 and 8043.
4. Add `SSLPassword="changeit"` (or your keystore password value) to the 8043 and 8443 connectors.

Note:

If you plan to change your keystore password from the default value of `"changeit"` to some other value, then follow the steps given at Optional Task: Change the Default Value Keystore Password.

1. If already installed, stop your Manager, open the

   `DS_DATA/conf/installation.properties` file, and set `DSConfigureOnStartup=true`.
2. Start the Manager, and perform a Manager Install. On the **Local Configuration**step, you must see your new HTTPS values. Make sure to enable SSL. Complete the install and restart the Manager.
3. If you have already installed Drivers from this GridServer installation, you must edit their `driver.properties` files to use the new HTTPS URL before they use SSL. Engines reconfigure themselves to use the new secure reinstallation; the Director URLs in all Engine Configurations change to https://host:sslport.

Task C Configure the Manager

1. In the Administration Tool, go to **Admin > System Admin > Manager Configuration > Security**.
2. You can set any or all of the SSL parameters on this page to true.
3. To configure the keystore passwords, change the passwords under the **Miscellaneous**heading.
4. Under **HTTPS Communication**, set **SSL Port**to 8043.
5. Restart the Manager.

Task D Configure the Clients

1. If you already installed Drivers and Engines from this TIBCO GridServer® installation, you must change their properties files to point to the new HTTPS URL before they use SSL. Configure the following:

   - Drivers: Driver HTTPS
   - Engines and Engine Daemons: Engines and Engine Daemon HTTPS

Optional Task: Change the Default Value Keystore Password

You must do the following changes before you make changes in the certificate and restart the Manager:

- Change the value in the Administration Tool at **Admin > System Admin > Manager Configuration > Security > Miscellaneous > Manager Keystore Password**(for `manager.keystore`) and **SSL Keystore Password**(for `server.keystore`).

- In the `server.xml` file, add the `keystorePass="yournewpassword"` parameter to the 8043 and 8443 connectors.
- When reconfiguring the Manager (Primary Director and Broker, or Primary Director) to SSL Broker (Standalone or Failover), under the `DS_DATA/conf` location, you must make the following additional changes:

  a. Rename `director.properties` to `broker.properties`.

  b. Edit `broker.properties` and make the following changes:

  - Replace `"/messageserver[@name\="Director"]"` with `"/messageserver[@name\="Broker"]"`
  - Remove the following line, if it exists:

    `/messageserver[@name\="Director"]/plugin[@class\="DirectorInfoPlugin"]/property[@name\="AuthenticateManager"]=False`

Driver HTTPS

The following files are used for Driver HTTPS:

- JDriver: `ssl.keystore` — Includes the self-signed DataSynapse certificate plus CA certs. This is not needed if you are using a CA-signed certificate. The Driver looks for this in the same directory as `driver.properties`, or you can set this location with the API. It can also be placed in the classpath or working directory.
- .NET Driver: `ssl.crt` — This is `ssl.pem` renamed so that Windows recognizes it. This is not needed if you are using a CA-signed certificate. If you do, double-click on the cert to install it.
- C++ Driver: `ssl.pem` — On a UNIX machine, it might not be needed if using a CA-signed certificate, since most distributions place CA certs in locations known to OpenSSL. It is required by Windows. The Driver looks for this in the working directory.
- Python Driver: Copy the updated `ssl.pem` file to the `./GridServerSDK-Linux/examples/service/python/client` folder.

Engines and Engine Daemon HTTPS

To enable Engines and Engine Daemons to trust the Broker:

1. Place the `ssl.pem` and `ssl.keystore` files in the DS_DATA`/engineUpdate/shared` directory.

2. Copy both files to the root directory of each Engine installation. You must copy the `ssl.pem` and `ssl.keystore` files by hand to each Engine already installed, and to any subsequently installed Engines.
3. When installing Engines, you must use the SSL port, 8043 and for Windows use `https` in the URL. For UNIX, you must give the `-l y` arguments to use SSL.

   You must enable HTTPS on the Manager for login, connection for Engine Daemons, and connection for Engine instances.

   To enable SSL for Engine instance and Engine Daemon login, you must set the Managers to the HTTPS location in the Engine Configuration.

   To enable SSL for Engine communication, you must enable it on the Manager. SSL is enabled for Engine Daemons on Directors. If your Manager is configured to use HTTPS for all Messaging, Engines already use HTTPS.

   To enable SSL for Engines if you did not enable HTTPS for all messaging:
4. Go to **Admin > System Admin > Manager Configuration > Security**.
5. Under **Component****HTTPS Communication**, set **Use HTTPS for Engine Communication**to True.
6. Click **Save**.

   To enable SSL for Engines Daemons on a Manager:
7. Go to **Admin > System Admin > Manager Configuration > Security**.
8. Under **Component HTTPS Communication**, set **Use HTTPS for Engine Daemon Communication**to True.
9. Click **Save**.

To enable domain-only authentication, you can use a wildcard (*) for the hostname in the CN field of your certificate. (For example. your CN could be set to `*.example.com`.)

Broker and Director SSL

The communication between Manager components (such as the connection between Brokers and Directors, and the connection between Primary Director and Secondary Director) can be configured to use SSL. Note that because they use pure sockets for communication, HTTPS does not need to be enabled on the Manager.

If you are enabling SSL for Manager components, it must be enabled on **all**Managers.

Note:

You must not change your Primary Director until all Brokers and Secondary Director have been changed.

To change to SSL connections on a Manager:

1. Copy `server.keystore` into the `DS_DATA/certs` directory of your Directors.
2. If using a self-signed certificate, copy `ssl.keystore` into the `DS_DATA/certs` directory of your Brokers and Secondary Directors.
3. Go to **Admin >****System Admin >****Manager Configuration > Security**, and update the SSL KeyStore Password on all Directors.
4. Perform a Manager Reinstall
5. In the **Local Configuration**step, set SSL Enabled to true.
6. Under Broker and Secondary Director Login, set SSL Enabled on Directors to true.

SSL On Large Grids

The Apache Tomcat AJP Connector can support many keep-alives (KA); however, the KA timeout is set to a default value of 60 seconds to reduce load on the Manager. This means that any time a new connection is necessary, the client must perform an SSL handshake to establish the HTTPS connections, which can impact performance.

Drivers poll on a regular basis, so they can maintain their connections. However, Engines can lose their connections when working on long tasks if the heartbeat is longer than the KA timeout. Idle Engines also lose connections on large grids where the Engine Wait time is longer than the KA timeout.

If you require HTTPS for Engines and you have a large grid with many Engines, you might notice Broker performance issues. Consider increasing this timeout to alleviate this issue. However, you must also monitor Broker performance, as this increases the number of connections that the AJP connector must poll.

Keep-alive is configured in Engine Configurations, located in the Administration Tool at **Grid Components > Engines > Engine Configurations.**The property to change is **File Server > Keep Alive Time (seconds)**.

Configuring a Reporting Database

The GridServer Manager can use a reporting database to store tables containing statistics for Services, tasks, Engines, and other records.

Reporting Database Requirements

To configure a reporting database, you must have a supported third-party database. For a list of supported databases, see the readme file included with GridServer.

Configuring the Reporting Database

To configure the reporting database, first install your Manager; second, configure the database; and third, configure the Manager to use the new database. Then, restart the Manager.

Install the Manager

1. Install and configure your GridServer Manager. Restart the Manager after completing the installation.
2. Ensure that the `tables.<db>.sql` file needed by the `createdb` script is in the `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db` directory.

Configure the database

1. Set the `CLASSPATH` on the Manager containing the Director that communicates with the database. The `CLASSPATH` must include the database driver JAR files applicable to your database. For example, to create an SQL Server database in a UNIX shell:

   ```
   export CLASSPATH=JDBCDriver/mssql-jdbc-13.4.0.jre11.jar
   ```

   replacing *JDBCDriver* with the actual path to the directory containing your driver JAR files.
2. Run the `createdb` script provided in `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db` directory on your Manager. Pass the name of the database properties file as an argument. Before you run the script, change the property file settings to match those of your database.

> **Note:**

- Use the Microsoft JDBC Driver for SQL Server (`mssql-jdbc` version 7.0 or later) for both the `createdb` step and the runtime connection. If you are using a Java 11 runtime, use the jre11 build included in version 7.1.1.
- The `JDBCDriver` parameter specifies the path to the directory that contains the driver JAR file. Using the same driver version ensures consistency across the `createdb` process, the `com.microsoft.sqlserver.jdbc.SQLServerDriver` entry in the connection table, and the Active Directory (AD) authentication.

Configure the Manager

1. Log in to the GridServer Administration Tool with an account that has a Security Role with the **Manager Reconfigure**feature enabled.
2. Go to **Admin****> System Admin > Manager****Configuration**> **Database**.
3. Configure the values in the **Reporting Database Connection**section using the appropriate table below. Note that *hostname* is the hostname and *database* is the name of the database.

Oracle

PROPERTY	VALUE
URL	`jdbc:oracle:thin:@` *hostname* `:1521/` *database*
Driver	`oracle.jdbc.OracleDriver`
Transaction Isolation	`TRANSACTION_READ_COMMITTED`

Microsoft SQL Server

PROPERTY	VALUE
URL	`jdbc:sqlserver://` *hostname* `:1433;DatabaseName=` *dbName* `;SelectMethod=` (`SelectMethod=cursor` is required)
Driver	`com.microsoft.sqlserver.jdbc.SQLServerDriver`
Transaction Isolation	`TRANSACTION_SERIALIZABLE`

PostgreSQL

PROPERTY	VALUE
URL	jdbc:postgresql://*hostname*:5432/*dbName*
Driver	org.postgresql.Driver
Transaction Isolation	`TRANSACTION_READ_COMMITTED`

1. To configure Microsoft SQL Server using AD authentication, please refer to Configuring Active Directory (AD) Authentication for the SQL Server Reporting Database (On-Premise).
2. Copy the corresponding JDBC driver's JAR file into the Manager's `DS_MANAGER/webapps/livecluster/WEB-INF/lib` directory.
3. Restart the GridServer Manager.

> **Note:**To retrieve the reporting database `username` and `password` from HashiCorp Vault instead of entering them manually, refer to Configuring HashiCorp Vault for Reporting Database Credentials.

Configuring Active Directory (AD) Authentication for the SQL Server Reporting Database (On-Premise)

GridServer Manager by default, connects to your SQL Server reporting database using SQL Authentication (SQL login and password). However, you can configure GridServer Manager to use Windows Active Directory (AD) authentication to securely connect to an MS SQL reporting database. This authentication method uses on-premise Active Directory credentials over NTLM (NT LAN Manager), through the Microsoft JDBC driver.

When you use NTLM, you do not require a Service Principal Name (SPN) or Kerberos configuration. The GridServer Manager host does not even need to be joined to your company's Windows domain. You simply need to provide the AD username and password in the reporting-database connection settings.

> **Note:**

- This authentication method applies only to the Manager's database connection. This authentication mechanism is distinct from how users or other grid components authenticate to the GridServer environment as given in the "Using Windows Authentication and Authorization" section in the TIBCO GridServer® AdministrationGuide.
- Use this method for on-premise AD only.
- The driver's `authenticationScheme=NTLM` mode is required.
- Do not use `authentication=ActiveDirectory*` values. These values authenticate only against Microsoft Entra ID or Azure AD and require additional libraries (`msal4j`).
- The no-credential modes, such as `ActiveDirectoryIntegrated` or native `integratedSecurity` SSPI, are not supported because the Manager always opens the reporting connection with an explicit username and password.

Prerequisites

1. **JDBC Driver:**The Microsoft JDBC driver (mssql-jdbc 7.0 or later) is placed in `DS_MANAGER/webapps/livecluster/WEB-INF/lib`. NTLM support was added in version 7.0. For GridServer 7.1.1, use the jre 11 build. For example, `mssql-jdbc-13.4.0.jre11.jar`. No extra libraries are required for NTLM.
2. **Driver class:**The driver class `com.microsoft.sqlserver.jdbc.SQLServerDriver` (unchanged) is used. Do not use the legacy `msbase.jar`, `mssqlserver.jar`, or `msutil.jar drivers`, as they do not support AD authentication.
3. **SQL Server:**Windows Authentication (Mixed Mode or Windows-only) must be enabled on SQL Server.
4. **Account:**Use a dedicated AD service account, not a personal user account. The password must be managed and long-lived.

Procedure

1. To create the SQL Server login for an AD account, run the following commands:

   Note: The commands to be run by DBA (sysadmin) only once.

```
-- On master: create the server login for the AD account

CREATE LOGIN [DOMAIN\svc_gsreport] FROM WINDOWS;
```

```
-- On the reporting database: create the user and grant rights

USE [GridServerReporting];

CREATE USER [DOMAIN\svc_gsreport] FOR LOGIN [DOMAIN\svc_gsreport];


ALTER ROLE db_owner ADD MEMBER [DOMAIN\svc_gsreport];

-- Or least-privilege (schema upgrades then need a one-off elevation):

-- ALTER ROLE db_datareader ADD MEMBER [DOMAIN\svc_gsreport];

-- ALTER ROLE db_datawriter ADD MEMBER [DOMAIN\svc_gsreport];
```

2. To configure reporting database connection (NTLM), navigate to **Admin > System Admin > Manager Configuration > Database**.
3. Set the **Reporting DatabaseConnection**values as given in the following table.

Note: Ensure that you keep a back-up of the original values before making changes.

PROPERTY	VALUE
URL	`jdbc:sqlserver://hostname:1433;DatabaseName=dbName;SelectMethod=cursor;encrypt=true;trustServerCerti integratedSecurity=true;authenticationScheme=NTLM`
Driver	`com.microsoft.sqlserver.jdbc.SQLServerDriver`
Username	`svc_gsreport` (bare sAMAccountName without domain prefix)
Password	Password of the Active Directory (AD) account.

4. Restart the GridServer Manager.

The domain is provided in the URL (`domain=DOMAIN`). Keep the **Username**without domain. `SelectMethod=cursor` remains required. For a self-signed server certificate, set `trustServerCertificate=true`. For production, install a trusted certificate and set `trustServerCertificate=false`.

> **Note:**NTLM must be permitted in the environment. If the corporate policy (GPO) disables NTLM, this AD-authentication method is not available for the reporting-database connection.

Verifying the Reporting Database Connection

- **Manager log:**Check the Manager log (`DS_DATA/logs/server/gridserver.log`) for confirmation that the reporting database is set, with no SQLException or authentication errors.
- **SQL Server:**Confirm the scheme used on SQL Server by running the following SQL query:

```
SELECT s.session_id, s.login_name, c.auth_scheme

FROM sys.dm_exec_connections c JOIN sys.dm_exec_sessions s

ON c.session_id = s.session_id

WHERE s.login_name = 'DOMAIN\\svc_gsreport';
```

  The auth_scheme should read NTLM.
- **Functionality:**Generate reporting activity (connect an Engine, run a Service) and confirm that the new rows display in the reporting tables.

Reverting the Configuration

If authentication issues occur, you must revert the GridServer Manager's configuration changes. To revert the configuration, perform the following steps:

Procedure

1. Restore the original SQL-authentication URL (without `integratedSecurity`, `authenticationScheme`, or domain), Username, and Password.
2. Restart the Manager.

Configuring HashiCorp Vault for Reporting Database Credentials

The Reporting Database username and password, by default, reside within the Manager configuration. Alternatively, you can configure GridServer to retrieve these credentials from a HashiCorp Vault server at connection time. The vault provides only the username and password. You must still specify the reporting database `URL` and `Driver` in the **Reporting Database Connection**fields.

When you enable the vault, the primary Director connects to the Vault server to fetch the credentials. It caches them for a configurable duration, automatically refreshes them before expiration, and passes only the resolved username and password to the Brokers. Brokers never communicate with the Vault server and do not receive the Vault configuration.

Prerequisites

Before you configure GridServer to use HashiCorp Vault, ensure that you meet the following requirements:

- Verify that a HashiCorp Vault server (OSS or Enterprise) is online and reachable from the primary Director.
- A Key-Value (KV) version 2 (**KV v2**) secret exists in the Vault. This secret must contain exactly two keys: `username` and `password` that hold the reporting database credentials.
- Vault authentication details are available for either the **Token**or **AppRole**authentication method.
- Configure the **Reporting Database URL**and **Driver**fields in the Manager configuration. Vault does not provide these values.

> **Note:**If you enable Vault but leave **Reporting Database URL**and **Driver**fields blank, the reporting database will not set up, and GridServer will not contact the Vault server.

Store the Credentials in the Vault

Create a **KV v2**secret containing `username` and `password` data. For example, using the Vault CLI:

```
vault kv put secret/gridserver/reporting-db username=gs_report
password=<db-password>
```

If the **KV v2**engine is mounted at `secret`, use `secret/data/gridserver/reporting-db` as the API path in GridServer.

The `data/` segment is mandatory for **KV v2**secrets.

Configure GridServer

1. Log in to the GridServer Administration Tool. Ensure that your account has the `Manager Reconfigure` feature enabled in its **Security Role**.
2. Navigate to **Admin**> **System Admin**> **Manager Configuration**> **Database**.
3. In **Reporting Database Connection**, set the `URL` and `Driver` for your database type. For details, refer to Configuring the Reporting Database .
4. Set **Use HashiCorp Vault for DB credentials**to `True`.

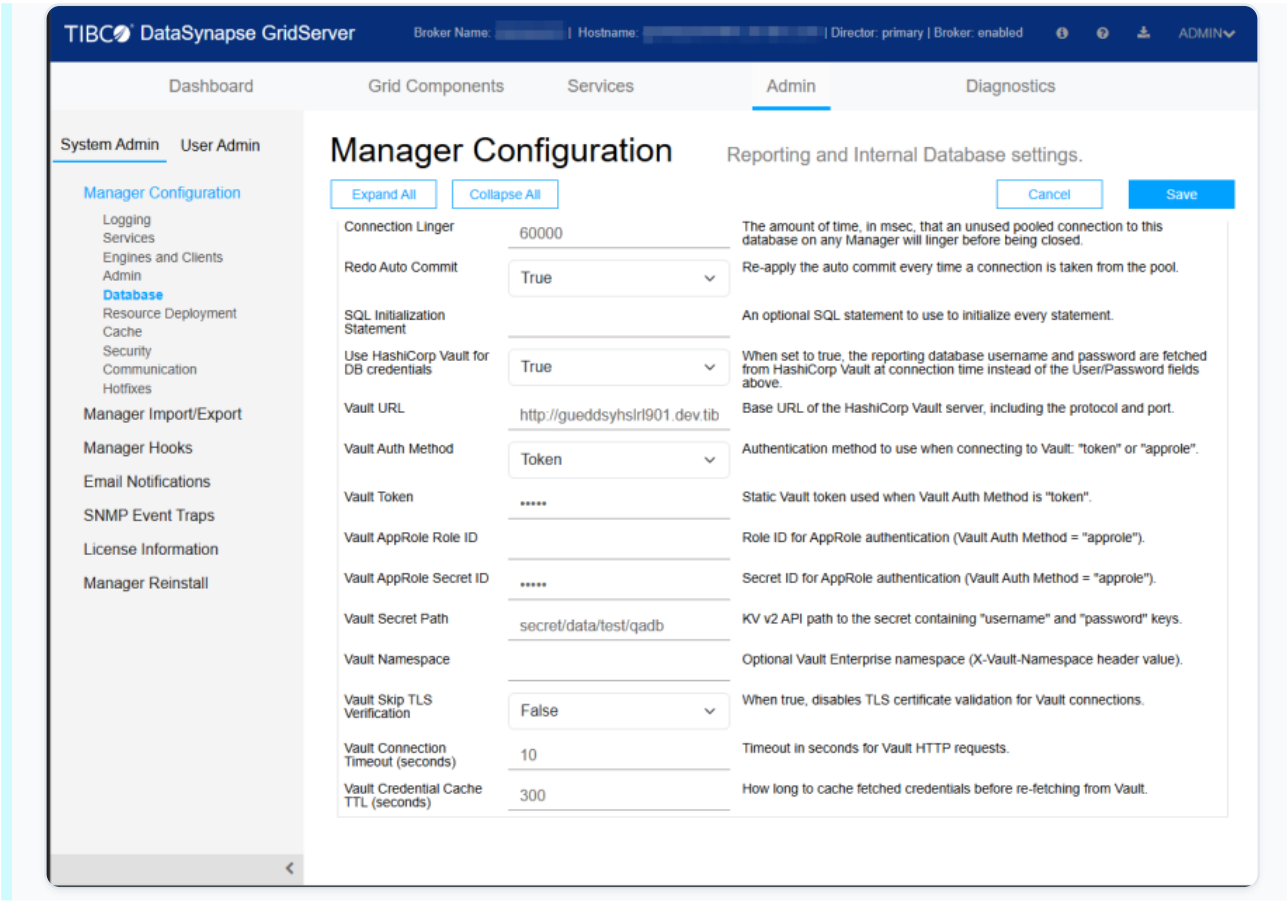
   The HashiCorp Vault configuration fields are available.
5. For your chosen authentication method, complete the HashiCorp Vault fields.

   For configuring the Vault fields, see Fields in Vault Settings.
6. Copy the appropriate `JDBC driver JAR` file into the Manager's `DS_MANAGER/webapps/livecluster/WEB-INF/lib` directory.

   > **Note:**This directory is the same as the standard reporting-database setup.
7. Restart the GridServer Manager.

   Vault settings take effect only after the restart completes.

> **Note:**When you enable Vault, leave the `User` and `Password` fields empty. The system ignores these fields and uses credentials from Vault.



Fields in Vault Settings

For configuring the fields in the Vault settings, navigate to **Admin**> **System Admin**> **Manager Configuration**> **Database**> **Reporting Database Connection**.

FIELD	DESCRIPTION	DEFAULT
Use HashiCorp Vault for DB credentials	When you set this field to `true`, the system fetches the reporting database `username` and `password` from HashiCorp Vault at connection time. The system does not use the **User**and **Password**fields. It still uses the `URL` and `Driver` fields. You must restart the system for this change to take effect.	`False`
Vault URL	The Base URL of the HashiCorp Vault server,	(empty)

FIELD	DESCRIPTION	DEFAULT
	which includes the protocol and port. For example, https://vault.example.com:8200.	
Vault Auth Method	The authentication method to use when connecting to Vault: `Token` or `AppRole`.	`Token`
Vault Token	The Static Vault token used when **Vault Auth Method**= `Token`. (Masked.)	(empty)
Vault AppRole Role ID	The Role ID for AppRole authentication (**Vault Auth Method**= `AppRole`).	(empty)
Vault AppRole Secret ID	The Secret ID for AppRole authentication (**Vault Auth Method**= `AppRole`).	(empty)
Vault Secret Path	The **KV v2**API path to the secret that contains the `username` and `password` keys. You must include the `data/` segment. For example, `secret/data/gridserver/reporting-db`.	(empty)
Vault Namespace	*Optional*. Vault Enterprise namespace. This value sets the `X-Vault-Namespace` header. For open-source Vault, leave this field blank.	(empty)
Vault Skip TLS Verification	If `true`, this option disables TLS certificate validation for Vault connections. Use this setting for development and testing environments only. In	`False`

FIELD	DESCRIPTION	DEFAULT
	production environments, this option must be `false`.	
Vault Connection Timeout (seconds)	The timeout, in seconds, for Vault HTTP requests.	`10`
Vault Credential Cache TTL (seconds)	The duration, in seconds, for which fetched credentials are cached before being re-fetched from the Vault.	`300`

Setting Vault Properties During Silent Installation

Alternatively, set the Vault properties in the `install.silent` response file. This file is in the Manager root directory, which is the same folder as `install.bat` or `install.sh`. During a silent installation, the silent installer reads this file. It writes the corresponding `DSVault*` values to `installation.properties` in the `DS_DATA/conf` directory. For more information, see Silent Installation.

The `install.silent` keys that correspond to the Reporting Database Connection Vault fields are in the following table.

`INSTALL.SILENT` KEY	CORRESPONDING FIELD	EXAMPLE
`vaultEnabled`	**Use HashiCorp Vault for DB credentials**	`true`
`vaultUrl`	**Vault URL**	https://vault.example.com:8200
`vaultAuthMethod`	**Vault Auth Method**	`token` or `approle`
`vaultToken`	**Vault Token**	`<token value>`
`vaultRoleId`	**Vault AppRole Role ID**	`<role id>`
`vaultSecretId`	**Vault AppRole Secret ID**	`<secret id>`
`vaultSecretPath`	**Vault Secret Path**	`secret/data/gridserver/reporting-db`

`INSTALL.SILENT` KEY	CORRESPONDING FIELD	EXAMPLE
`vaultNamespace`	**Vault Namespace**	*Optional*. `Vault Enterprise`

For example, a Token-auth configuration:

```
vaultEnabled=true

vaultUrl=https://vault.example.com:8200

vaultAuthMethod=token

vaultToken=<vault-token>

vaultSecretPath=secret/data/gridserver/reporting-db
```

Note:

- The settings `Vault Skip TLS Verification`, `Vault Connection Timeout (seconds)`, and `Vault Credential Cache TTL (seconds)` are not available in `installation.properties` file. These settings use default values of `False`, `10 seconds`, and `300 seconds`. You can modify these values in the Administration Tool. For additional details, refer to Securing Passwords and Internal.script.
- When you apply a hotfix or service pack to an existing Manager, the system does not automatically add new `DSVault*` keys to your `installation.properties` file. You must manually add these keys to the `DS_DATA/conf/installation.properties` file. Then, reinstall or restart the Manager to apply the settings.
- HashiCorp Vault storage is supported for the Primary Director only. Secondary Directors and failover configurations do not support this storage.
- The HashiCorp Vault integration does not cover either Engine Instrumentation or Scheduler Instrumentation. Vault-based credential resolution is implemented only for the Reporting Database path.

Authentication Methods

GridServer authenticates to Vault using one of the following methods:

- **Token**: You authenticate GridServer to Vault using the provided **Vault Token**. GridServer uses this token for all subsequent interactions with Vault.
- **AppRole**: GridServer logs in by sending an HTTP `POST` request to `<Vault URL>/v1/auth/approle/login` using the configured `Role ID` and `Secret ID`. GridServer then uses the client token returned by this login process. The system caches this token and reuses it until its lease nears expiration. You must ensure the `Role ID` and `Secret ID` are correctly configured for AppRole authentication.

Credential Refresh Process

The system caches credentials for the duration specified by the **Vault Credential Cache TTL (seconds)**and refreshes them proactively before the cache expires. If a database connection fails due to an authentication error, GridServer invalidates the current cache and retries with newly fetched credentials. Rotated credentials are only applied to new connections. Connections already established and present in the connection pool continue to utilize their existing credentials until those connections are re-established.

Security Notes

- You must set **Vault Skip TLS Verification**set to `False` in production environments to validate the Vault server's certificate. If you enable this setting, the Manager log records a security warning.
- The UI masks the **Vault Token**and **Vault AppRole Secret ID**fields.
- Brokers do not receive the Vault configuration or long-lived credentials. The Director provides only the resolved `username` and `password`.

Configuration Verification

After the Manager restarts, verify that the reporting database connects successfully. On a successful connection, the Manager log records the following `INFO`-level messages:

```
Fetching reporting DB credentials from Vault at <Vault URL>

Successfully fetched reporting DB credentials from Vault path: <Vault
Secret Path>
```

Troubleshooting

SYMPTOM / MESSAGE	LIKELY CAUSE
Log warning that Vault is enabled but the Reporting Database `URL` and/or `Driver` is empty. The reporting database is not set up and Vault is not contacted.	The `URL` and/or `Driver` are not set. Vault supplies only the `username` / `password`. Set the `URL` and `Driver` as well.
`Vault auth method is 'token' but ... Token is not configured`	**Vault Auth Method**is `Token`, but **Vault Token**is empty.
`Vault auth method is 'approle' but ... Role ID / Secret ID is not configured`	`AppRole` is selected, but the `Role ID` or `Secret ID` is empty.
`... Secret Path is not configured`	**Vault Secret Path**is empty.
`Could not extract 'username' and/or 'password' keys from Vault secret at path: ...`	The secret does not contain keys named exactly `username` and `password`, or the path is not a **KV v2** `data/` path.
`Vault returned HTTP <status>: ...`	Vault rejected the request (for example, bad token/role, wrong path, insufficient permissions, or the server is unreachable).

Deploying With Openshift

GridServer supports the use of the OpenShift platform for deployment and management of GridServer components in containers. The GridServer SDK provides example files to deploy Engines in OpenShift.

Introduction

GridServer supports the use of the OpenShift container platform for deployment and management of GridServer components in containers. The GridServer SDK provides example files to deploy Engines in OpenShift.

About OpenShift

OpenShift is an application platform built for running and managing containers using Kubernetes. OpenShift handles application lifecycle management functionality, such as automated building and deployment.

Deployment can be run in different models, including products for hosted public cloud, private cloud, or using on-premise infrastructure.

For more information about OpenShift, see https://www.openshift.com/.

About Kubernetes

Kubernetes is an open-source container. It handles the deployment, scaling, and maintenance of Docker containers. OpenShift uses Kubernetes for the container orchestration and management layer of its platform, using Red Hat Enterprise as the container host, and adding the application lifecycle management layer.

For more information about Kubernetes, see https://kubernetes.io/.

About the GridServer Containerization

The GridServer SDK includes an example Dockerfile which shows how to containerize a GridServer Engine to run in Docker. This enables you to deploy GridServer engines to the Openshift platform.

This example is in the SDK in the `examples/container/gridserver` directory. The `Dockerfile` is in the `engine` directory.

Deploying the GridServer Engine Dockerfile into OpenShift

The following procedure deploys an Engine to the OpenShift environment with the OpenShift CLI using the `oc` command.

Prerequisites

- This process assumes you are familiar with OpenShift. See the OpenShift web site at https://www.openshift.com/ for more information.
- Obtain an OpenShift account and login credentials from the OpenShift web site.
- Install the OpenShift CLI. This is available from the Red Hat customer portal after you have created an account.
- Download the GridServer SDK from the GridServer Administration Tool.
- Install git on your local workstation.
- Add the Engine Dockerfile to your git repository.

  It's also suggested to add all the folders under the `container` directory of the GridServer SDK into the git repository.

Procedure

1. Log in to OpenShift with following command:

```
oc login
```

2. Create a new project:

```
oc new-project ProjectName --description="your description"\

  --display-name="your display name"
```

   For example:

```
oc new-project gs-engine --description="GridServer Engine \

   Container Deployment" --display-name="GridServer Engine"
```

3. Create a new application:

```
oc new-app your_dockerfile_git_repository --build-env \

   GS_ARCHIVE_URL=location_of_Engine_archive_file \

   DIRECTOR_URL=location_of_GridServer_Primary_Director

   ENGINE_CONFIG=Engine_Configuration
```

For example:

```
oc new-app https://git.example.com/tibco/gridserver.git \

   --context-dir=engine --build-env \

   GS_ARCHIVE_URL=http://192.168.0.100:8080/livecluster/public_html/
register/install/unixengine/DSEngineLinux64.tar.gz \

   DIRECTOR_URL=192.168.0.100:8080 ENGINE_CONFIG=default
```

Note that if the GridServer Engine is running in an OpenShift cluster with its internal IP address and is not visible by the Driver or Manager, you need to create a route in the OpenShift project, and then configure the File Server URL in the Engine Configuration.

For example: create a route in OpenShift project as:

```
    http://gs-engine.gs-engine.cluster-proxy-host
```

and then set **Engine Configuration > File Server > File Server URL**to http://gs-engine.gs-engine.cluster-proxy-host.

This routes external communication from the GridServer Driver and Manager to the Engine internal file server (the default port is 27159).

After Deployment

After deployment, you can perform application lifecycle management tasks with the OpenShift CLI. See the OpenShift documentation for more details.

Configuring Scheduler Instrumentation

The GridServer Manager can use the `scheduler_info` table, which contains details of GridServer Scheduler operations.

Scheduler Instrumentation Requirements

To configure a database for Scheduler Instrumentation, you must have a supported third-party database. For a list of supported databases, see the readme file included with TIBCO GridServer®.

To configure Scheduler Instrumentation

To configure Scheduler Instrumentation, you must perform the following steps:

1. Install the GridServer Manager and GridServer Engine
2. Task A Configure the Database
3. Task B Configure the Manager (to use the database)
4. Restart the Manager (The JDBC JAR file is picked up by the Manager only after a restart.)

Task A Configure the Database

Create the `scheduler_info` table in any supported database (Oracle, PostgreSQL, MSSQL). Refer to the syntax provided in the `scheduler-info-readme`, available in the `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db` directory.

Task B Configure the Manager

1. Log in to the GridServer Administration Tool with an account that has a Security Role with the **Manager Configuration Edit**feature enabled.
2. Go to **Admin > System Admin > Manager Configuration > Services > Scheduler Tracker**.
3. Configure the values in the **Scheduler Tracker**section on Broker by using the appropriate values given in the following table:

   Note that the `hostname` is the name of the host and `dbName` is the name of the database.\

PROPERTY	VALUE
Oracle	
Database URL	jdbc:oracle:thin:@*hostname*:1521*/dbName*
Database Driver	oracle.jdbc.OracleDriver
Microsoft SQL Server	
Database URL	jdbc:sqlserver:// *hostname*:1433;DatabaseName=*dbName*; SelectMethod=cursor (SelectMethod=cursor is required)
Database Driver	com.microsoft.sqlserver.jdbc.SQLServerDriver
PostgreSQL	
Database URL	jdbc:postgresql://*hostname*:5432*/dbName*
Database Driver	org.postgresql.Driver

4. Copy the corresponding JDBC driver's JAR file to the following Manager's directory: `DS_MANAGER/webapps/livecluster/WEB-INF/lib`.

5. Restart the Gridserver Manager.

To view the data, you require an account that has Security Role with the Scheduler Instrumentation feature enabled. The data can be viewed from the UI at **Diagnostics > Scheduler Instrumentation**on the Broker.

Level 1

Level 2

Level 3

Configuring Engine Instrumentation

The GridServer Manager can use the `engine_ins` table, which contains details of the Engine balancing process.

Engine Instrumentation Requirements

To configure a database for Engine Instrumentation, you must have a supported third-party database. For a list of supported databases, see the readme file included with TIBCO GridServer®.

To Configure Engine Instrumentation

To configure Engine Instrumentation, you must perform the following steps:

1. Install the GridServer Manager and the GridServer Engine
2. Task A Configure the Database
3. Task B Configure the Manager(to use the database)
4. Restart the Manager (The JDBC JAR file is picked up by the Manager only after a restart.)

Task A Configure the Database

Create the table `engine_ins` in any supported database (Oracle, PostgreSQL, MSSQL). Use the syntax provided in the `engine-ins-readme` available in the `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db` directory.

Task B Configure the Manager

1. Log in to the GridServer Administration Tool with an account that has a Security Role with the Manager Configuration Edit feature enabled.
2. Go to **Admin > System Admin > Manager Configuration > Services > Engine Instrumentation**.
3. Configure the values in the **Engine Instrumentation**section on Primary Director by using the appropriate values given in the following table:

   Note that the `hostname` is the name of the host and `dbName` is the name of the database.\

PROPERTY	VALUE
Oracle	
Database URL	jdbc:oracle:thin:@*hostname*:1521*/dbName*
Database Driver	oracle.jdbc.OracleDriver
Microsoft SQL Server	
Database URL	jdbc:sqlserver:// *hostname*:1433;DatabaseName=*dbName*; SelectMethod=cursor (SelectMethod=cursor is required)
Database Driver	com.microsoft.sqlserver.jdbc.SQLServerDriver
PostgreSQL	
Database URL	jdbc:postgresql:*//hostname*:5432*/dbName*
Database Driver	org.postgresql.Driver

4. Copy the corresponding JDBC driver's JAR file to the following Manager's directory: `DS_MANAGER/webapps/livecluster/WEB-INF/lib`.
5. Restart the GridServer Manager.

To view the data, you require an account that has Security Role with the Engine Instrumentation feature enabled. The data can be viewed from the UI at **Diagnostics > Engine Instrumentation**.

On the Director

Here, you can see two Brokers: `1358717965` and `1956710939` and related information.

On the Broker

Here, the Broker `1358717965` contains only the data specific to it.

Type	Value	Time
Engine Logoff	Broker : 1358717965 Logging off 1 engines for balancing [rkabra-x1e-3]	2021/02/01 21:57:43.771
Engine Logoff	Broker : 1358717965 Logging off 1 engines for balancing [rkabra-x1e-3]	2021/02/01 22:33:44.144
Engine Logoff	Broker : 1358717965 Logging off 3 engines for balancing [rkabra-x1e-10, rkabra-x1e-1, rkabra-x1e-9]	2021/02/10 16:09:28.586
Engine Logoff	Broker : 1358717965 Logging off 3 engines for balancing [rkabra-x1e-8, rkabra-x1e-2, rkabra-x1e-3]	2021/02/10 16:12:23.861
Engine Logoff	Broker : 1358717965 Logging off 1 engines for balancing [rkabra-x1e-4]	2021/02/10 16:56:19.780

Introducing TIBCO GridServer

- GridServer Overview
  - Product Overview
  - Grid Services
  - GridServer Architecture
  - Operating Environment
- GridServer Services
  - Services
    - Clients
    - Service Implementation
    - Service Session
    - Service benefits
  - Binary-level Integration
- Managing Engines
  - Engine Routing and Balancing
    - Balancing and Service Discriminators
    - Engine Weight-Based Balancer
    - Home/Shared Balancer
    - Engine Balancer Configuration
    - Engine Upper and Lower Bounds
  - Failover Brokers
  - Example Use Cases
    - N+1 Failover with Weighting
    - Engine Localization with Sharing
  - Engine Configuration

- Editing an Engine Configuration
- Creating a New Engine Configuration
- Copying an Engine Configuration
- Setting the Engine Configuration Used by Engines
- Setting the Director Used by Engines
- Configuring Engines With Multiple Network Adapters
- Configuring Engine Daemons to Use SNAT
- Using the System Classloader on an Engine

- Configuring a Global Shared Grid Library Directory
- Configuring When Engines Run
  - Manual Mode
  - Auto Mode
- Configuring How Many Engines Run
- Running Engines in Multiplexed Mode
  - Communication and Task Scheduling
  - Configuration
- Configuring 64-bit Engine Daemons to run 32-bit Services
  - Configuration
  - Specifying that a Service is win32
  - Routing 32-bit Tasks to 64-bit Engines
- Configuring a Caching HTTP Proxy Server
- Configuring an External Engine Daemon Admin Tool
- Quarantine Brokers
  - Quarantine Broker Concepts
  - Quarantine Status on Engines
  - Requirements
  - Configuring a Quarantine Broker
  - Setting Quarantine Status on Engines

        - Quarantine Broker Constraints
- Inside the GridServer Director
    - Director Roles
        - Routing
        - Authentication
        - Load Balancing
    - Configuration Management
    - Database Management
    - Resource Management
- Inside the Broker
    - Broker Roles
        - Scheduling
        - Engine and Driver Management
        - Communication
    - Resource Management
- Inside GridServer Topology
    - Redundancy
    - Volume
    - Topography
    - Direct Data Transfer
- GridCache
    - Overview
    - General Capabilities
        - API
        - Modes
        - Cache Configuration and Access
        - Data Storage

        - Attributes
        - Consistency/Synchronization
        - Cache Loaders
        - Cache Loader Write-through and Bulk Operations
        - Notification
        - Disk/Memory Caching
        - Cache Region Scope
        - Data Conversion Matrix
    - Using The GridCache API
    - Fault Tolerance and GridCache
- Inside the Client
    - Client Types
    - Installation and Configuration
- Inside the GridServer Batch Processing Facility
    - Parts of a Batch
        - Batch Definitions
        - Batch Components
    - Scheduling Batch Definitions
    - Administering Batches

GridServer Overview

This guide is your complete introduction for learning about TIBCO GridServer® Manager concepts. It includes a complete overview of GridServer fundamentals and is meant to be read first, before installation or development.

Product Overview

GridServer is a highly scalable software infrastructure that allows application services to operate in a virtualized fashion, unattached to specific hardware resources. Client applications submit requests to the Grid environment and GridServer dynamically provisions services to respond to the request. Multiple client applications can submit multiple requests in parallel and GridServer

dynamically creates multiple service instances to handle requests in parallel on different Grid nodes. This architecture is therefore highly scalable in both speed and throughput. For example, a single client sees scalable performance gains in the processing of multiple requests, and many applications and users see scalable throughput of the aggregate load.

A scalable architecture provides linear gains in performance with the addition of more hardware resources, while making no impact on the manageability of the distributed application platform. This means that the system must be as manageable with ten nodes as it is with a thousand nodes. GridServer offers this type of operating environment—incremental resources added to the system impact performance in a linear fashion, but have no effect on management—including deployment of application services; configuration and administration; runtime management and monitoring; and historical reporting and usage statistics.

This section is a high-level overview of the GridServer platform. It covers three separate areas: Grid Services, the GridServer architecture, and management and operations of the infrastructure.

Grid Services

The ability to integrate a new technology into a variety of legacy environments is a core requirement of any strategic technology. GridServer offers support for multiple languages and different integration strategies to meet this goal.

With GridServer, Grid clients (applications that use the Grid) can interoperate with any Service hosted on the Grid—regardless of the language or system that the components are running. For example, an application can make Grid Service calls that are implemented in any language and system, including the following systems:

GRID SERVICE IMPLEMENTATIONS
Java Classes
.NET Classes
C++ Classes
C Functions in a DLL/SO
Excel (extension package)
Any executable or script
R functions

Any Service can be accessed by Grid client applications in a number of ways, including the following ways:

GRID CLIENT INTERFACES
Java/J2EE
.NET API (VB.NET, C#)
COM Object
C++/C
Batch Clients
R functions

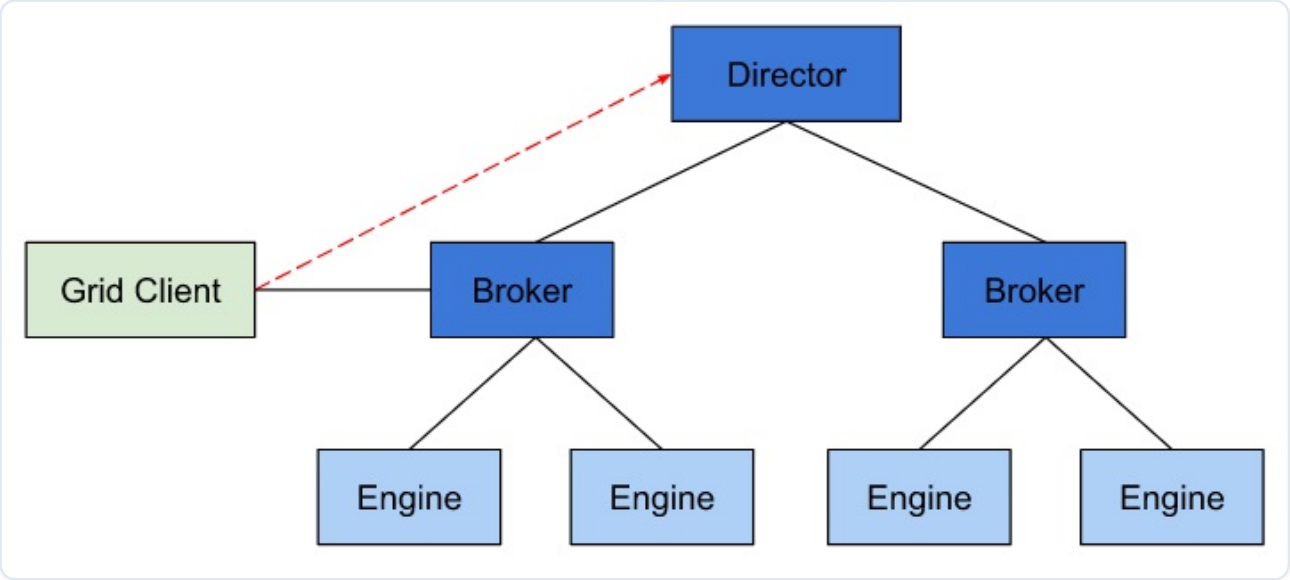
GridServer enables a truly Service-oriented approach to Grid computing. The Grid client creates a Service instance and calls operations (methods) on the Service. These calls are virtualized and invoked on multiple Grid nodes. This virtualization provides limitless scalability for hosting these Services and offers a simple programming model for invoking these Services.

Services are described in more detail in GridServer Services.

GridServer Architecture

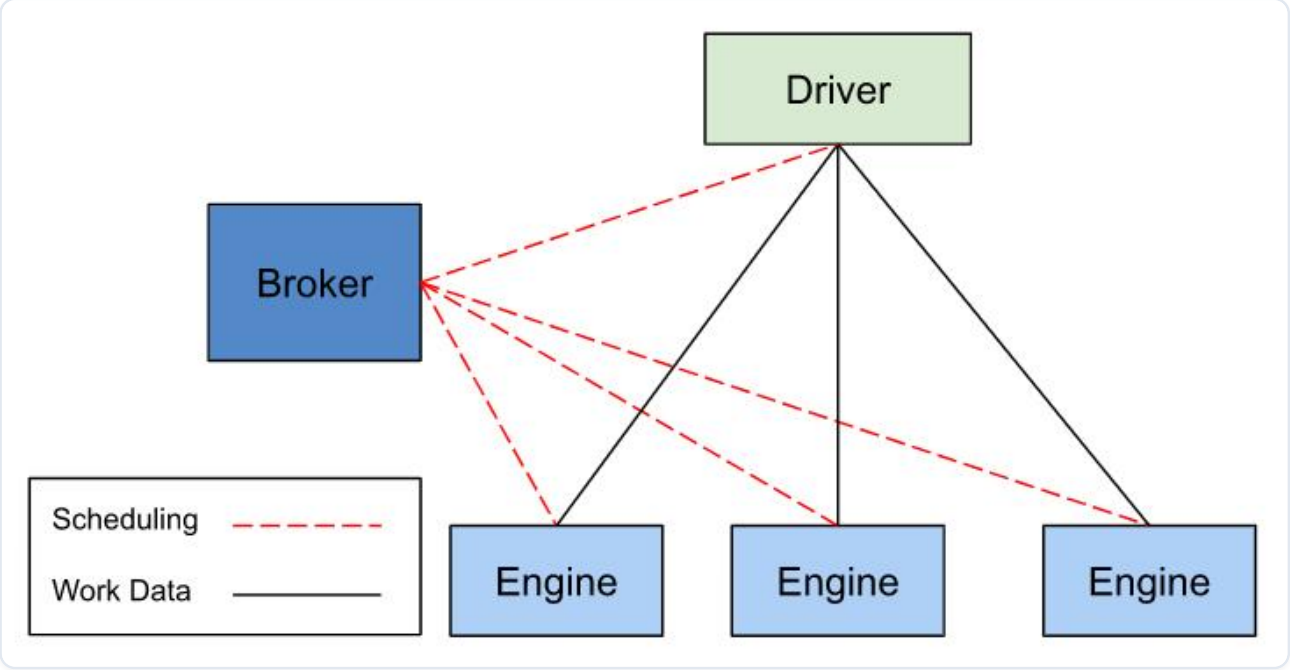
DataSynapse GridServer has four different components in its architecture:

- **Grid Clients**— The components that submit service requests to the Grid. Also called Drivers.
- **Engines**— The processes that host and run services on the Grid nodes.
- **Brokers**— The component that provides request queuing, scheduling and load-balancing. Brokers are also responsible for managing Engines.
- **Directors**— The component that assigns Grid Clients and Engines to Brokers based on policy. They manage and load balance Engines across available Brokers.



Engines and Grid Clients log in to the Director and are authenticated; the Director then routes the Engines and Grid Clients to available Brokers.

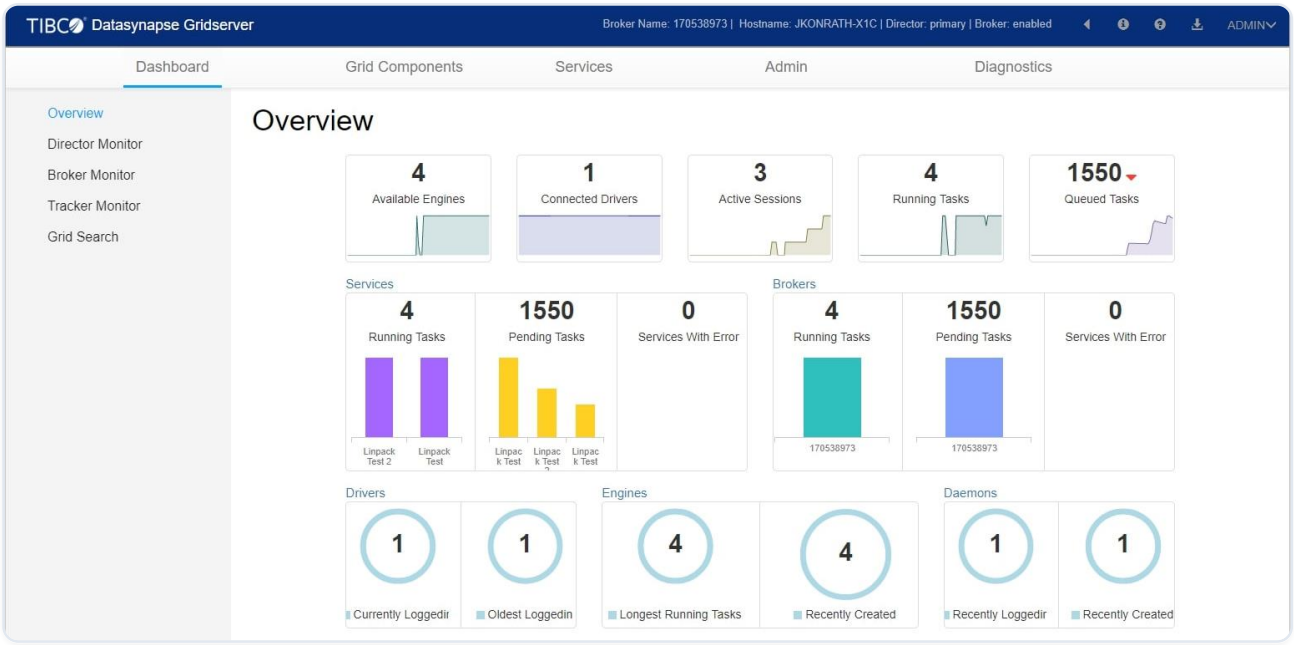
Grid Clients submit requests through the Broker. Engines receive work requests from the Broker, and in most cases, exchange data directly with the Grid Clients. This allows the system to be highly scalable. Since the Brokers manage all work requests, load balancing is optimal and resilience is built into the system.



Brokers manage Engines and Drivers, and schedule work with lightweight messages. Drivers and Engines exchange work data directly when using Direct Data Transfer.

Operating Environment

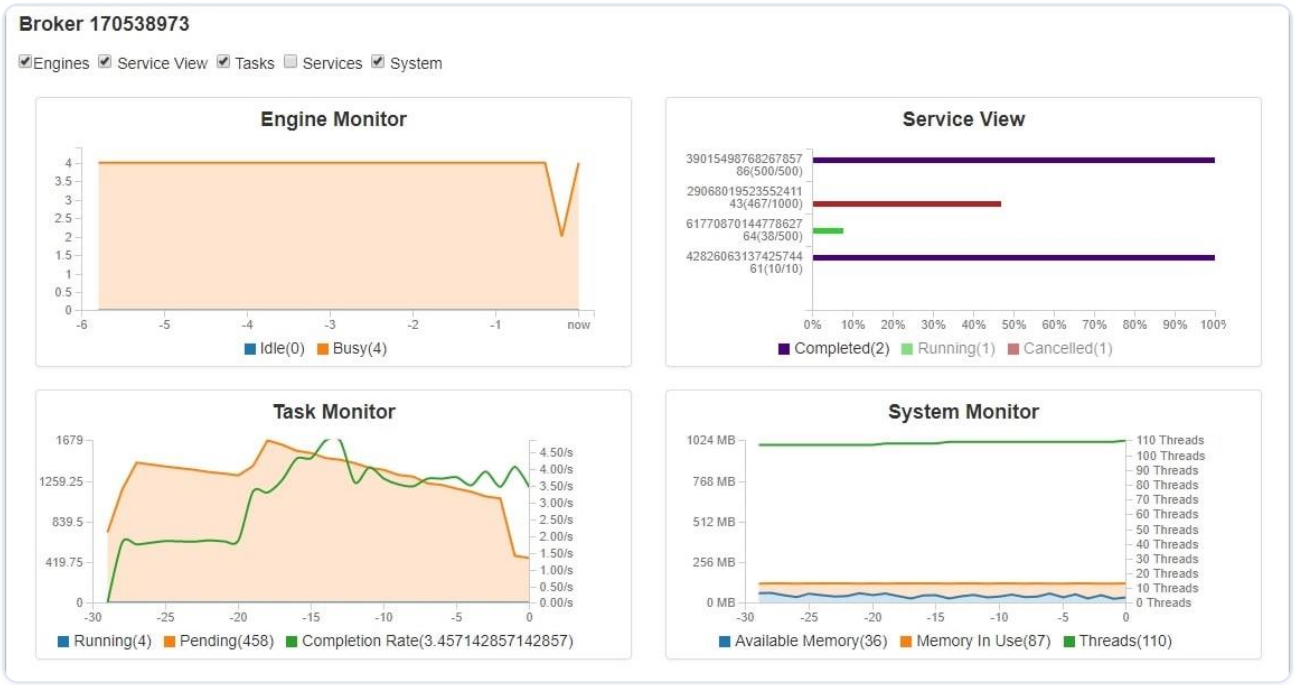
GridServer includes a highly manageable operating environment featuring a web-based console, the GridServer Administration Tool, and a SOAP-based administration interface. Both allow you to deploy Services, manage the workloads running on the Grid, and configure the GridServer environment. The screenshot below shows the home page in the GridServer Administration Tool.



GridServer Administration Tool

Deploying and registering Services and applications is made available throughout the Administration Tool. Uploading binaries and controlling versions is straightforward.

The Administration Tool has a number of graphical runtime monitors to quantify Grid usage and current operations. Diagnostic tools enable remote debugging, logging, and error extraction.



The Broker Monitor

Management of the GridServer platform is described in more detail in the TIBCO GridServer® Administration.

GridServer Services

A Service is a self-contained application or business function that is distributed to remote computing resources. A client application makes requests, which are routed by GridServer to those resources, which then return results to the client. One use of a Service is to load balance an application, whereby the client application receives requests and submits them asynchronously to the Service, which then executes these multiple, unrelated computations in parallel. Another use is to execute a single, larger computation. The client splits the computation into multiple, independent pieces, submits them asynchronously, and combines the returned results.

Services

The Service-Oriented method of defining work in GridServer is a standards-based model. It uses a thin client model, which promotes easy integration of an existing implementation. It also promotes language interoperability, as clients written in different languages can invoke methods in Service Implementations written in the same or other languages.

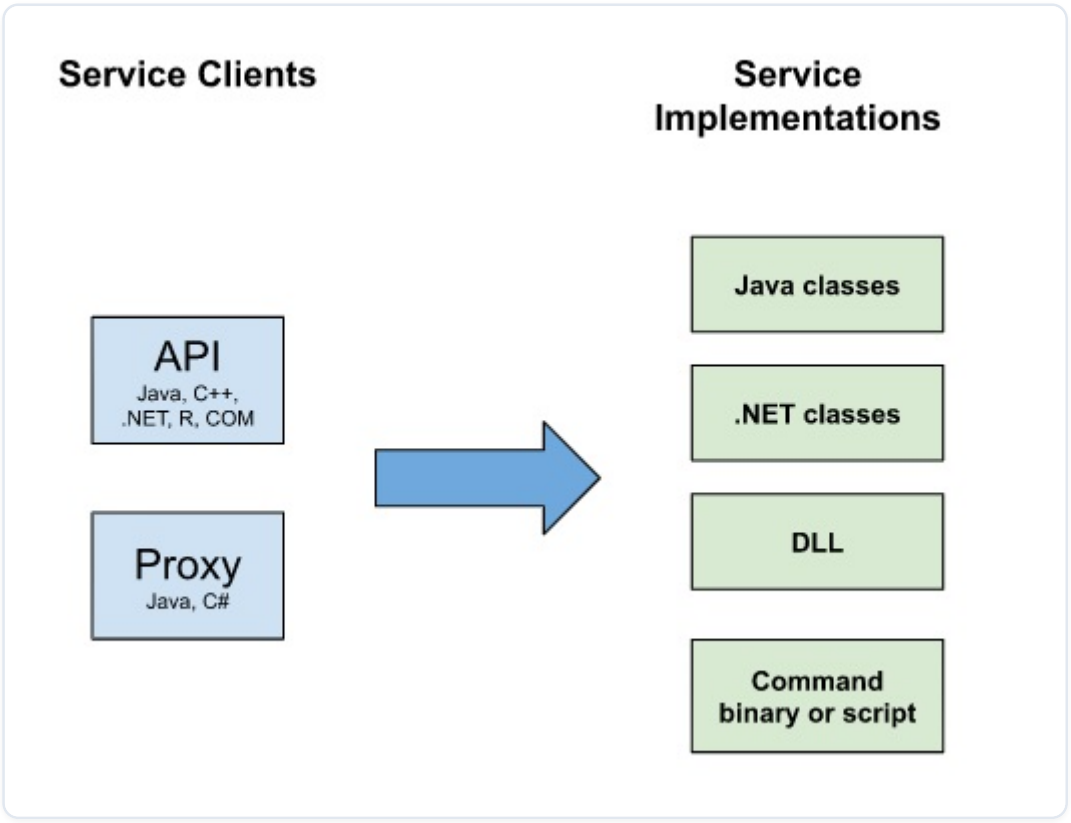
The Service-Oriented method uses two components: Clients and Service Implementations. Descriptions of both follow.

Clients

A Service Session is created on a client for a given named Service Type. The client invokes methods using implementation resources that are distributed on Engines.

You can create a Service client in different ways:

- A client-side API in Java, COM, C++, R, Python, or .NET.
- A service proxy of Java or .NET client stubs generated by GridServer.



The relationship between Service Clients and Service Implementations

Service Implementation

Service Implementations are libraries or executables that are deployed to Engines and that process requests from clients. They process data and return results back to the client. Service Implementations are associated with Service Types by registering them on the Director. When a client makes a client request, it sends the request to a Manager instead of directly requesting an Engine to do the work. This one-to-many relationship provides fault tolerance and scalability for Services.

Service Implementations can be constructed with any of the following options:

- Arbitrary Java classes
- Arbitrary .NET classes

- A Dynamic Library (.so, .DLL) with methods that conform to a simple input-output string interface
- R functions
- A command, such as a script or binary executable

Integration as a Service in most cases requires minimal changes to the client application.

Service Session

A running service is a *Service Session*. This includes the Service Client, Service Implementation, and Service state on all components. After a client creates a Service and the Service Implementation is running on Engines, this is collectively called the Service Session.

Service benefits

There are many advantages to Services:

- **Cross-language**— Client and Service can be in different languages
- **Dynamic**— Method names can be determined dynamically, or use generated proxies for type safety
- **Flexible**— Use synchronous or asynchronous invocation patterns; can use client proxies generated by GridServer
- **Virtual**— Client-Engine correspondence is not one-to-one; Service requests are adaptively load balanced
- **Stateful**— Despite being virtual, stateful Services can be handled
- **Standards**— Standards-compliant

For more information about Services, see the *TIBCO GridServer® Developer's Guide*.

Binary-level Integration

PDriver, or the Parametric Job Driver, is a Driver that executes existing command-line programs as a parallel processing Service using the GridServer environment without using the API, taking full advantage of the parallelism and fault tolerance of GridServer.

PDriver achieves parallelism by running the same program on Engines several times with different parameters. A script is used to define how these parameters change. For example, a distributed search mechanism using the `grep` command could conduct a brute-force search of a network-

attached file system, with each task in the Service being given a different directory or piece of the file system to search.

PDriver uses its scripting language, *PDS*, to define Services. PDS scripts can also set options for a PDriver Service, such as remote logging and exit code checking.

For more information about PDriver, see the *TIBCO GridServer® Developer's Guide*.

Managing Engines

This section contains information for managing GridServer Engines. For information about installing Engines, see the TIBCO GridServer® Installation. For information about troubleshooting Engine issues, see" Diagnosing Engine Issues" in the TIBCO GridServer® Administration.

Engine Routing and Balancing

Engines are dynamically allocated resources. They can migrate among Brokers based on criteria such as load and policy. Use the Engine Balancer to manage logins and re-route Engines to maintain an optimal balance across the grid. The Engine Balancer is a component of the Director. The Primary Director’s balancer always runs. The Secondary Director’s balancer runs only if the Primary is down.

The Director handles routing and load balancing as follows:

1. The Director regularly polls Brokers for the states of Engines on the Brokers. The Director tests routing mechanisms against each Engine and determines the optimal location for each Engine. Changes in the states of Engines due to load balancing requirements result in changes in the optimal distribution of Engines on Brokers.
2. The Director sends a request to each Broker that has Engines that must be moved, to log those Engines off.
3. Engines that must return to their home Broker log off immediately, regardless of the task timeout setting.
4. Shared Engines that are busy restart immediately without finishing the current task. Engines that are not busy log off immediately.
5. After an Engine logs off or restarts, it then logs in to the optimal Broker.

Two balancer algorithms are available. Choose one according to how you plan to use the grid:

- The **weight-based**balancer algorithm attempts to distribute Engines equally by relative weights, and it also allows rule-based routing using Engine properties.

- The **Home/Shared Balancer**routes Engines based on an Engine's assigned Home Brokers, and the sharing policy of Home Brokers to other Brokers. Both balancers take into account the number of running and pending tasks on each Broker, and the desired maximum and minimum number of Engines for each Broker.

If you change the Engine Balancer on the Director, you must restart it. Also, all balancer settings must be equal on Primary and Secondary Directors. You can configure routing settings for online or offline Brokers.

Balancing and Service Discriminators

When the Director polls Brokers for Engine information, it also collects information about Service discriminators and blacklisted Engines. The Director avoids a situation where Engines are at their home Broker and can't take Services due to Service discrimination or blacklisting, but also won't be shared to another Broker. (Task-level discriminators are not taken into account.)

All of the following conditions must be true for a Service, to report a Service discriminator:

- The Service is not complete.
- The Service has pending tasks.
- The number of busy Engines working on the Service does not exceed the max Engines option of the Service.

To limit CPU and network usage during balancing, the maximum number of discriminators reported by each Broker can be configured on the Director at **Admin > System Admin > Manager Configuration > Engines and Clients > Max Service Discriminators**. This setting specifies the number of Services with discriminators per Broker that are considered.

If a Broker has more outstanding services with Service discriminators than the maximum specified, a message is logged in the Broker logs, similar to the following message:

```
INFO: [EngineSharing] Maximum number of discriminators 10 is reached when
collecting service level discriminators
```

When this occurs, the remaining discriminators over max (ten in the above example) is not considered when allocating Engines to this Broker. This does not prevent the Engines from being reallocated to the Broker even if the Engines don't satisfy the discriminators. If Max Service Discriminators is set to 0, no discriminators are considered.

If the Max Service Discriminators is set to a very high value, reporting of balance data from the Broker to the Director takes more time which slows down the balancing. If balancing is slowed down to an unacceptable level, lower the value of the parameter.

Engine Weight-Based Balancer

The Engine weight-based balancer allocates Engines to Brokers based on each Broker's Engine weight value. This value is the amount of Engines allocated to the Broker relative to the other Brokers' weights, when all Brokers are idle. The algorithm also considers session load and reallocates idle Engines to busy Brokers as they are needed. You can see a Broker's Engine weights value on the **Grid Components > Brokers > Broker Admin**page.

The Engine weight-based balancer permits rule-based routing through Engine Properties, when it is necessary to restrict some Engines to a set of Brokers. You can route Engines by their intrinsic properties, such as `cpuTotal`, and by user-defined properties. Create and assign user-defined properties with the **Grid Components > Engines > Engine Properties**page. Use the **Grid Components > Brokers > Broker Routing**page to set up routing rules based on these properties.

Home/Shared Balancer

The Home/Shared Engine balancer uses an algorithm in which an Engine has a set of Home Brokers that it always works on while it has outstanding tasks, yet the Engine can be shared to other Brokers when there are no outstanding tasks on any home.

The balancer uses Broker needs and Engine preferences for Brokers to allocate Engines to Brokers. Each Engine divides the existing Brokers into tiers (unordered sets of Brokers). The two default tiers are:

- The Engine's home Brokers
- The shared Brokers of those home Brokers

  You can introduce a third tier by splitting shared Brokers into two groups: preferred shared brokers, and common shared Brokers. The higher the tier, the more the Engine prefers the Brokers in that tier.

  The balancer uses the following rules:

- An Engine is routed to the highest-tiered Broker that has pending tasks. If multiple Brokers in the same tier have pending tasks, the choice is made at random, as if all weights were 1. The number of Engines moving to a Broker is capped at the number of pending tasks if there are more pending tasks than available Engines in the tier.

- An Engine leaves its current Broker only if there is a needy Broker in a higher tier. An Engine does not move to a lower-tiered Broker unless it is idle.
- Failover Brokers are never allocated Engines unless they have pending tasks.
- If `Options.MAX_ENGINES` is set on a Service and the number of Engines from other Brokers are not shared with this Broker to run the Service. If there are more Engines on a Broker than the sum of all `MAX_ENGINES` values across active Services, the excess Engines are reported as available for sharing.

Use the GridServer Administration Tool to configure Brokers. Configure an Engine's home Broker with the **Grid Components > Engines > Engine Configurations**page. Configure Broker tiers (which Brokers share Engines with other Brokers) with the **Grid Components > Brokers > Broker Configuration**page as follows:

1. To set the first tier of Brokers, fill in the **Preferred Broker Sharing**field. Supply a comma-delimited list of the Brokers with which the current Broker shares Engines.
2. To set the second tier of Brokers, fill in the **Common Broker Sharing**field. Supply a comma-delimited list of other Brokers with which the current Broker shares Engines.
3. You can also define additional tiers of Brokers by delimiting them with semicolons in the **Common Broker Sharing**field. For example, to enter a second and third tier of Brokers, you could enter the list B1, B2, B3; C1, C2, C3.

For example, an Engine configuration's home Brokers are A and B. A's preferred broker is C; its common list is D,E. B's preferred Broker is F; its common list is G. An Engine with this configuration uses the following preferences: first: A, B; second: C, F; third: D, E, G. Within each tier, Brokers are equal, and ordering doesn't matter.

You can also use the Admin API to get or set the tiers. In `com.datasynapse.gridserver.admin.BrokerAdmin`, use the methods `setSharedBrokers` and `getSharedBrokers` to set or get the tier string. You can also create a Batch Definition that uses the Admin API to change the tiers according to a time schedule. For more information about using the Admin API, see the TIBCO GridServer® Developer's Guide.

On the **Grid Components > Brokers > Broker Configuration**page, you can also set a minimum number of idle home Engines for a Broker by adding the **Min Idle Home Engines**property column to the page. If the idle home Engine count is below this value, home Engines (idle or busy) are not logged off or shared to other Brokers.

Engine Balancer Configuration

To configure Engine Balancing on the Director, go to **Admin > System Admin > Manager Configuration > Engines and Clients**, and change the following properties:

SETTING	DESCRIPTION
Engine Balancer	The Engine balancer to use: **Weight-Based**or **Home/Shared**.
Rebalance Interval	The amount of time, in seconds, between balancing episodes. (Previously called the Poll Period.)
Logoff Timeout	The amount of time in seconds that an Engine waits to finish a task before logging off.
Broker Query Timeout	The maximum time to wait for Broker's reply for balancer data queries. The value is in milliseconds.
Soft Logoff	If true, Engine logoffs do not restart the JVM unless needed by a home Broker. This enables them to retain state and log in faster.
Engine Fraction	The fraction of extra Engines that moves to another Broker on a balance. This can be set to less than 1 to dampen Engine movement. For example, if the fraction is 0.5 and the balancer determines that a Broker has eight extra Engines, it moves four on the first balance. Assuming those Engines move, on the next balance it determines that there are four extra and moves two, and so on.
Engine Balance Maximum	The maximum number of Engines that can move to another Broker on a rebalance. The maximum applies over the entire grid. For example, if this property is set to 100 and the balancer determines to rebalance 200 Engines (after taking Engine Balance Fraction into account), then only 100 Engines are actually rebalanced.

SETTING	DESCRIPTION
Max Service Discriminators	The maximum number of service discriminators to consider when reporting balance data from each Broker.
Allow Routing When Sharing	If Broker Routing properties are used when the Sharing balancer is enabled. Under most circumstances, when using the Sharing balancer, it is best not to also use routing.
Treat Pinned Engines as Busy	If pinned, Engines are reported as busy from a sharing point of view. In some use cases, treating pinned Engine as busy Engines can reduce Engine fluctuation and improve overall grid performance. Note that this setting affects Engine sharing only. For information about Engine Pinning, see the TIBCO GridServer® Developer's Guide.

These settings must be identical on all Directors.

Engine Upper and Lower Bounds

You can configure upper and lower bounds on the number of Engines that can be logged in at a given time. Set these upper and lower bounds on the **Grid Components > Brokers > Broker Admin**page. If the columns for bounds are hidden, add them using the **Column**control. The minimum value specifies that the balancer algorithm always leaves at least this amount of Engines (assuming there are this many) on the Broker unless the Engines are needed by Brokers in higher sharing tiers. The maximum value is the cap on the total amount of Engines to allow on the Broker. The balancing algorithms use both values.

Failover Brokers

A Failover Broker temporarily takes over executing Service Sessions when the Client has no other Brokers to which it is permitted to connect. From the Client perspective, Failover Brokers become part of the pool of active Brokers when there are no other non-Failover Brokers on which the client is permitted. From the Engine perspective, a Failover Broker becomes part of the active pool when there are active sessions in progress on that Failover Broker. In either case, the algorithm now views this Broker as a non-Failover. It is important to take Failover Brokers into account when setting up the routing configuration. For example, if you are setting up a role to allow a client on

only one Broker under normal conditions, you must also include a Failover Broker in its Manager List if you wish this client to have a failover if its main Broker goes down.

For more information, see "Grid Fault-Tolerance and Failover" in the TIBCO GridServer® Administration Guide.

Example Use Cases

This section describes example use cases of client routing and load balancing.

N+1 Failover with Weighting

An organization has four groups using all available Engines in a grid. One group has a guaranteed allocation of at least half of the grid any time it needs it, and the other three groups share the remaining Engines.

- **Brokers** Set up five Brokers. Each group gets a Broker, plus one is used for failover.
- **Drivers** Create four roles, one for each group. In each role, set the **Manager List**value to the group's Broker and the failover Broker. Assign the roles to the appropriate users.
- **Engines**Use the weight-based Engine Balancer. Adjust **Engine Weight**on the Broker Admin page so the first group's Broker is weighted at 3.0, and the other three groups' Brokers are weighted as 1.0. Set the failover Broker weight at 1.0, so that a group is not assigned any more resources than normal if their Broker goes down.

Engine Localization with Sharing

A company has two groups, one in New York and one in London. Each has a single middleware application that has a Driver that connects to its own Broker. Each group also has a set of CPUs that it expects to always be working on their own calculations. However, there are times when one group's Broker is idle, so they are allowed to share with each other.

- **Brokers** Set up four Brokers, a regular and a failover for each group. Each regular Broker shares with the other regular Broker, plus its own failover Broker.
- **Drivers** Create two roles, one for each group. In each profile, set the **Manager List**value to the group's Broker and its failover Broker. Assign the roles to the middleware application user.
- **Engines** Use the Home/Shared Engine Balancer. Set up two Engine Configurations, "London" and "New York," that set the Engines' homes to their respective Brokers.

In this scenario, the application always connects to its local Broker, unless it is down, in which case it moves to its failover. Whenever that Broker has pending requests, all of its Engines are always

local. If the other group's Broker is idle, or if it does not need all of its Engines, any of its idle Engines are routed to the Broker that needs it.

Engine Configuration

Engine and Engine Daemon behaviors are controlled by a centralized profile called an *Engine Configuration*. When new Engine Daemons are installed, they use the default Engine Configuration for their platform (Windows, Linux, Solaris, or other supported platforms.) Individual Engine instances take the configuration of their Engine Daemon. You can edit existing Engine Configurations, or create new Engine Configurations for different subsets of Engines on your grid.

Editing an Engine Configuration

To change an Engine's settings (for example, to point it to a new Director), you edit the Engine Configuration used by the Engine. Settings change on all Engines using that Configuration.

> **Note:**
>
> Changing an Engine Configuration causes Engines using that Configuration to restart.

> To change values in an Engine Configuration:

1. In the GridServer Administration Tool, go to **Grid Components > Engines > Engine Configuration****s**.
2. Select a configuration in the Configuration list. The list contains names of all default Engine Configurations, plus any custom configurations that have been created.
3. Change any of the values that appear in the Engine Configuration.
4. Click **Save**to keep your changes, or **Cancel**to revert changes.

> **Note:**
>
> When adding values to the Environment Variables box, it is possible to set variables, particularly `PATH`, that can cause an Engine to fail to start.

Creating a New Engine Configuration

In some situations, you might need a subset of Engines to behave differently than other Engines on your grid. For example, you might want a large group of Engines to report to a different Director, or use a different compiler runtime. Instead of individually configuring each Engine, you can create a new Engine Configuration with different Director settings, and assign the new Configuration to a subset of Engines.

To create a new Engine Configuration:

1. Go to **Grid Components > Engines > Engine Configuration****s**, and click **Add**.
2. Select a platform for the Engine Configuration from the list.
3. Next to the selected platform, enter a name for the Configuration.
4. Click **Create**.

The initial values of the new Engine Configuration are the same as the default Configuration for the selected platform. You can then change the values of the configuration and click **Save**.

Copying an Engine Configuration

In a situation where you need several similar Engine Configurations, it's often faster to make a copy of an existing Engine Configuration, rather than creating many individual Configurations.

To copy an existing Engine Configuration:

1. Go to **Grid Components > Engines > Engine Configuration****s**.
2. Select an existing Engine Configuration from the list.
3. Click **Copy**. You are prompted for a name for the new Configuration.

Setting the Engine Configuration Used by Engines

To change the Engine Configuration of an Engine Daemon:

1. Go to **Grid Components > Engines > Daemon Admin**.
2. In the **Configuration**column, select an Engine Configuration from the list next to an Engine Daemon.

   To change a large number of Engine Daemons' Configuration at once:

3. Go to **Grid Components > Engines > Daemon Admin**.
4. Use the **Results Per Page**and **Search**controls to limit the table to only the Engine Daemons you want to change. For example, enter win32 in the search box and select **OS**as the column

to search, then click **Go**. If you have more than twenty win32 Engines, enter 100 in the **Results Per Page**box before you do your search. The goal is to get all of the Engine Daemons you want to change on one page.

5. In the **Actions**list, select **Configure Daemons on Page**.
6. The **Engine Daemon Editor**page opens, displaying all of the Engine Daemons previously in the table.
7. Select an Engine Configuration from the **Configuration**list.
8. Click **Save**.

Setting the Director Used by Engines

The Primary and Secondary Directors for an Engine is set during Engine installation. You can later change the Directors to which an Engine reports, by changing the Engine Configuration used by the Engine.

To configure an Engine's Directors:

1. Go to **Grid Components > Engines > Engine Configuration****s**.
2. Select the Engine Configuration used by the Engine. This is typically the operating system of the Engine.
3. In the **Directors and Brokers**section, change **Primary Director URL**and **Secondary Director URL**to the corresponding addresses and ports of the Primary and Secondary Directors, in the format `http(s)://address:port`.

Note that this changes the Directors for all Engines using that Engine Configuration. Also note that when moving Engines from one grid to another, Engines lose any custom Engine properties that are defined on the former Director's grid, because those are stored in the grid's administrative repository.

Configuring Engines With Multiple Network Adapters

In some network configurations, the host running an Engine can have more than one physical or logical network interface. When the Engine starts, it checks what IP address is in use and uses it to advertise its file server location. In cases where there is more than one network interface, it's possible for the Engine to pick the wrong one. To remedy this it is possible to force the Engine to choose a specific interface. To configure the Engine to use a different network interface, select the Engine Configuration it uses on the Engine Configuration page, and set the **Net Mask**value under the **File Server**heading to match the network range on which to run the Engine.

Configuring Engine Daemons to Use SNAT

You can configure Engine Daemons for use on the other side of a NAT from the Manager and Drivers. To do this, set the environment variable `DS_USE_SNAT_IP_ADDRESS` on an Engine Daemon to the IP address that you want it to report to the Manager and all Drivers as its address. Additionally, you must set **Self Ping**to `FALSE` in the Engine Configuration.

Using the System Classloader on an Engine

In most situations, the default classloader is used to load client classes. In some instances however, some applications might require the use of the system classloader. This requires deploying extra files to enable the Engine to use the system classloader.

To use the system classloader:

1. Copy the `DS_MANAGER/webapps/livecluster/WEB-INF/etc/DSEngine.jar` file to `DS_DATA/engineUpdate/shared`.
2. Go to **Grid Components > Engine > Engine Configuration****s**, select an Engine Configuration, and change the value of **Classloader**to **System**.

Note that this requires an Engine restart for any Grid Library that contains a `jar-path`.

Configuring a Global Shared Grid Library Directory

You can configure Engines to use a shared directory for Grid Libraries instead of Engines downloading their own copies.

To configure a global shared Grid Library directory:

1. Disable Engine synchronization on the **Admin > System Admin > Manager Configuration > Resource Deployment**page.
2. On the **Grid Components > Engines > Engine Configurations**page, set the **Grid Library Path**to a shared directory with read-only access that is available from all Engines.
3. Unpack all necessary Grid Libraries into the shared Grid Library directory, with each library in its own subdirectory matching archive filename.
4. Deploy the corresponding Grid Library archives to the Broker. The Broker still requires the same access to the Grid Libraries it always had, though Engines can now access them with an alternate method.

Notes:

- Grid Libraries do not automatically update when using this deployment method. You must manually unpack and update the shared Grid Libraries, and restart the Engines.
- The shared Grid Library directory must contain both the Grid Library archive files and the extracted archives.
- Permissions are not maintained in `.zip` archives regardless of Broker OS.
- Permissions might be maintained when using `.tar.gz` or `tgz` Grid Libraries.
- If the configuration does not maintain permissions in any container or distribution Grid Libraries, you must make sure the executable permissions are set properly for any executed scripts.

> **Note:**
>
> Using a global shared Grid Library directory with multiple Engines using the same network share can cause a significant drop in performance.

Configuring When Engines Run

You can configure GridServer to avoid conflicts between work and regular use of the machine. This is called adaptive scheduling, which you configure to adapt the Engine's activity level to your computing environment. The following section describes the methods used to configure when an Engine Daemon runs Engines. Unless otherwise indicated, all of the configuration options are available in Engine Configurations, which are modified on the **Grid Components > Engines > Engine Configurations**page.

Manual Mode

When Manual Mode is enabled in an Engine Configuration, the Engine Daemon runs at all times. This is the default. You can change an individual Engine Daemon to auto mode, or clear the **Enabled**option under **Manual Mode**in the Engine Configuration page to switch all Engines using that configuration to Auto Mode.

Auto Mode

Auto Mode lets you specify if Engines run on a computer, based on its utilization. For example, you might not want to run Engines when someone is logged in to the computer, or if its CPU utilization is above a certain percentage.

For Windows Engines, there are two methods for determining utilization: **User Idle**and **Processor Utilization**. When you select Auto Mode, select one of the two options. The default is User Idle.

Auto Mode for UNIX Engines always uses Processor Utilization mode. In addition, you can also limit the hours an Engine runs using the **Restricted Hours**settings. Each setting is described below.

User Idle

The User Idle feature is available for Windows Engines only. User Idle starts Engines if mouse and keyboard input have been idle for a given time on the computer hosting the Engine Daemon. This time is entered in seconds, and the default is 600 (10 minutes). Engines halt when there is mouse or keyboard input.

User Idle also has a percentage setting, which is used to determine when Engines stop running, in addition to UI activity. If processor utilization (in percent) on the host computer exceeds this value, Engines stop. To disable this feature, set it to 100%, the default.

Processor Utilization (Windows Engines)

The Processor Utilization option enables Engine Daemons to monitor system CPU usage, and start or stop Engines based on this statistic.

To use this option, enter two values: a percentage and a number of seconds. When CPU utilization drops below the given percentage for the given number of seconds, Engines start. For example, by default, Engines start when CPU utilization is below 50% for more than 30 seconds. When utilization goes above the same CPU threshold value for the specified number of seconds, Engines stop.

CPU usage on single CPU systems is calculated as the total CPU usage, minus the CPU overhead of the Engine. Note that this does not include CPU usage for the Engine, invoke, or CPU idle processes. In multi-CPU systems where an Engine Daemon launches multiple Engine instances, CPU utilization is calculated independently for each CPU.

In multi-CPU systems, all Engine instances start and stop incrementally. This is called **Incremental Scheduling**, and is the default. When a CPU threshold value is reached, Engines start or stop one at a time. After an Engine starts or stops, there is a delay for a configurable interval—by default, the interval is 10 seconds. Utilization is checked again on the next CPU after the interval delay, and the process repeats.

If you do not select Incremental Scheduling, Engine Daemons use **Non-Incremental Scheduling**. In this case, all Engines on an Engine Daemon start or stop at the same time.

Processor Utilization (UNIX Engines)

On UNIX Engines, the Processor Utilization option works similar to its Windows counterpart, but there are some specific changes:

- You can configure both a starting and a stopping threshold and time. This lets you, for example, start Engines at 40% CPU utilization but stop them at 50%. (The Engines must also be above or below the CPU utilization percentage for the specified period of time.)
- The CPU usage sampling is averaged over a configurable period of time, with the default at 10 seconds.

When using incremental scheduling, Engines are started only when all other Engines are busy. This results in fewer Engine restarts when starting a new Service that requires a restart to load Grid Libraries, and restarts due to download of new libraries when idle.

Also, in non-incremental scheduling, CPU utilization is the average CPU utilization across all CPUs, and not individual CPU utilization. This total CPU utilization percentage is calculated by adding the CPU utilization for each CPU and dividing by the number of CPUs.

For example, if a four-CPU computer has one CPU running at 50% utilization and the other three CPUs are idle, the total utilization for the computer is 12.5%. Likewise, if the maximum CPU threshold is set at 25% on a four-CPU machine and four Engines are running, and a non-Engine program pushes the utilization of one CPU to 100%, all four Engines exit. Even if the other three CPUs are idle, all Engines still exit. In this example, if the minimum CPU threshold is set at 5%, all four Engines restart when total utilization is below 5%.

This only applies to UNIX Engines, when non-incremental scheduling is used. With incremental scheduling on UNIX Engines, and in all scheduling on Windows Engines, each Engine Daemon only looks at the CPU utilization for the CPU on which it is running.

Similar to Windows, CPU utilization calculation does not include CPU usage for the Engine, invoke, or CPU idle processes.

Excluding Processes From Utilization Calculations

Windows and Linux Engines can also be configured to exclude a list of processes from utilization calculations.

The **Exclude the following list of processes from the processor utilization calculation**property in Windows and Linux Engine configurations can be used to specify processes that are ignored in utilization calculations. The value of this parameter must be a comma

or semicolon-delimited list of case insensitive names, and are typically the executable names without extension.

Any process spawned by a Service must be in this list to prevent the Service from shutting down the Engine.

Note that Engine, invoke, and CPU idle processes are already excluded from utilization calculations by default.

Restricted Hours

When using Auto Mode, you can also specify a range of hours when Engines run. For example, if you want Engines to only run from 9:00 AM to 5:00 PM daily, configure this using Restricted Hours. You can also have GridServer ignore restricted hours settings on weekends.

To configure Restricted Hours:

1. Go to **Grid Components > Engines > Engine Configuration****s**.
2. Either select an existing Engine Configuration from the list to modify, or create a new profile.
3. Under the **Restricted Hours**heading, select the first checkbox.
4. Enter a time range when the Engines are allowed to run. For example, if you want Engines to run from 9:00 AM to 5:00 PM, enter 9:00 and 17:00.
5. Select the second option to ignore Restricted Hours settings on Saturdays and Sundays.

Restricted Hours settings do not apply to Manual Mode.

Configuring How Many Engines Run

There are three settings in the Engine configuration that determine how many Engine instances an Engine Daemon can run.

The **SMP Enabled**property specifies if the Engine configuration starts an Engine instance per processor core on the machine. This is set to True by default. Specifically, when SMP is enabled, the Engine Daemon runs a number of Engine Instances equal to the value of the **Minimum Engine Instances**property or the number of processor cores on the machine, whichever is higher. By default, this is set to 1, meaning the number of Engine instances is always the number of cores.

If SMP is disabled, the Engine Daemon always runs the number of Engine instances set in the **Minimum Engine Instances**property in the Engine configuration.

You can also adjust the number of Engine instances used, to reserve a certain number of cores for non-Engine activity. The value of the **Core Relative Engine Instances**parameter is added to the value specified in **Minimum Engine Instances**. The relative value can be a positive or negative number.

For example, if you want to ensure that a single core is always available with no Engine running on it, set the value of **Core Relative Engine Instances**to -1. If a 4-core machine uses this Engine configuration, its number of Engine instances is the higher of its number of cores and the minimum instances (4) plus the relative instances value (-1), or 3. If a 2-core machine uses this Engine configuration, it starts one Engine instance, leaving the other core free.

You can also manually specify the number of instances that can run for an Engine Daemon. If you go to the **Grid Components > E****ngines > Daemon Admin**page, there is a list in the **Instances**column that enables you to select Default or auto SMP. You can type a number into this control to override the number of instances that can run.

Running Engines in Multiplexed Mode

On machines with multi-core processors, GridServer, by default, runs an Engine instance in its own process per logical CPU. While this enables each process to work on different Services, it also means each process runs in its own JVM, and requires its own set of application data. This can cause unwanted overhead when running the same Service across many CPU cores.

To address this issue, you can run Engines in multiplexed mode, meaning that all Engine instances run in a single process. This enables all Engine instances on that machine to share the same set of application data, and multiplex their communication with the Broker.

Multiplexed Engines appear as individual Engines in the GridServer Administration Tool and by the Admin API. Each multiplexed Engine has its own log, work, temp, and data directories. They produce the same output to the reporting database.

You can’t mix single-process Engine instances and multiplexed Engines on the same machine. For example, on an eight-core machine, you can’t have two processes with four multiplexed Engines each; you can only have eight multiplexed Engines in one process.

Multiplexed Engines have the following differences with Engines running in their own processes:

- All Engines are logged in to the same Broker at any given time. This is enforced by the balancer.
- All Engines work on the same Service session at any given time. This is enforced by the Scheduler

- All Engines reside in the same classloader, which means they can all share the same set of Service Session data.
- They share the same initialization data and Service object. The Service init method is only called once, and all updates are synchronized across all Engines.
- If one Engine has to log off, all Engines in the process are also logged off with the same reason.
- Incremental scheduling is not supported.
- Autopack is not supported.
- The Service implementation must be threadsafe. For more information about multiplexed mode development considerations, see the TIBCO GridServer® Developer's Guide.
- Running more than one task from a recursive Service is not supported.
- All multiplexed Engines share one set of properties. For example, getting the instance property always reports instance 0, regardless of the Engine instance.

When using multiplexed Engines, Engine resources are shared. The first Engine does all library loading, and this Engine is the first to start work. This also means that Engine Hooks are only loaded once, as hooks are loaded when Grid Libraries are loaded.

When using GridCache, there is one cache per process, and all communication is handled by Engine instance 0.

Communication and Task Scheduling

When Engines are configured as multiplexed, communication with the Broker is multiplexed over a single HTTP keep-alive session to reduce overhead. Also, only one heartbeat is communicated per process. Heartbeats are received by Engine instance zero and simulated to the other Engine Proxies.

To prevent starvation of Engines to other Service sessions, a lease duration is defined. This is the amount of time a set of multiplexed Engines work on one Service before they stop working on that Service and are allowed to work on others if necessary. The lifecycle for this is as follows, given a lease duration of 5 minutes:

- When work is started on the Grid, Engine instance 0 picks up a task.
- Immediately following that, all other instances pick up tasks from the same Service session.
- Assuming there is enough work, the instances continue to take tasks from that session.

- After five minutes has passed, when any instance finishes a task, it does not pick up any more tasks.
- As soon as the last instance has finished, immediately after the response has been written, Engine instance 0 requests another task from any appropriate session. It could be the same session or another session.

Configuration

To enable multiplexed mode, set the Multiplexed Mode property to true in the Engine configuration. The default value is false.

To set the lease time, go to **Admin > System Admin > Manager Configuration > Services**, and set the value of the `Multiplexed Engine Lease` property. The default time is one minute.

The `MULTIPLEXED_MASTER_ID` Engine property is set on any multiplexed Engine to contain the sessionID of Engine instance 0 within its process.

There are two buffer timeouts:

- `MULTIPLEXED_ENGINE_MAX_BUFFER_WAIT`: This is an absolute time set per Service.
- `MULTIPLEXED_ENGINE_AVG_BUFFER_WAIT`: This is a factor that is multiplied by the average task duration or the tasks executed during the lease. The default is 0.5. So if the average duration is 10 seconds, this timeout is 5 seconds.

The timeout is measured starting from the first send request.

Configuring 64-bit Engine Daemons to run 32-bit Services

The 64-bit Windows Engine Daemon can be configured to allow execution of 32-bit Services. If allowed, when an Engine takes such a task, it restarts, clearing all existing loaded libraries, and restarts as a 32-bit process.

Configuration

On the Windows Engine Configuration page, there is a setting called **Additional Platforms**under **Classes, Libraries, and Paths**. It is a list with two values, **None**and **win32**. By default it is **None**. Setting it to **win32**on a Windows 64-bit configuration enables the behavior; setting it on a Windows 32-bit configuration has no effect. This sets an Engine property named `additionalPlatforms` to `win32` or the empty string.

When this property is enabled, Engines download any Grid Libraries with the OS set to `win32` in the root element, in addition to all others it normally syncs.

Note that because Engines only take tasks if they have the root Grid Library of a Service downloaded, this inherently allows these Engines to take 32-bit Windows Services.

When this property is enabled, the following behavior occurs:

- When a 32-bit task is taken by a 64-bit Engine, it restarts as a 32-bit Engine, clearing all Grid Libraries except for the one from the Service and its dependencies
- When a 32-bit task is taken by a 32-bit Engine, it loads normally.
- When a task that is anything but 32-bit is taken by a 32-bit Engine, it restarts as a 64-bit Engine, clearing all Grid Libraries except for the one from the Service and its dependencies.
- When a task that is anything but 32-bit is taken by a 64-bit Engine, it loads normally.

Specifying that a Service is win32

Any Service for which the root Grid Library is marked as `os="win32"` is considered to be a win32 Service.

If a Service's root Grid Library is not marked as such, the Engine remains 64-bit and fails.

Routing 32-bit Tasks to 64-bit Engines

By virtue of the Engine having the 32-bit Grid Library, they by default are allowed to run such Services by the scheduler.

Note that existing applications might not use the OS attribute on the root Grid Library; rather they might use discrimination instead to route Services to win32 Engines. If you wish to use 64-bit Engine Daemons on these Services, you must set the OS attribute on that Grid Library.

Configuring a Caching HTTP Proxy Server

In a GridServer deployment where a Broker and its Engines are separated by a WAN, it can be inefficient to transfer the same data over the WAN to multiple Engines from the Broker or the Clients. One solution is to use an HTTP proxy server (such as Squid Web Cache) to cache the session’s init data, which any Engine that works on the session must transfer. You can specify a proxy server in an Engine configuration, and the proxy server caches the Service data for other Engines also using the same proxy server.

To use a proxy server such as Squid for resource synchronization or data transfer, configure **Proxy Host**and **Proxy Port**parameters to the proxy hostname and port. Two additional properties dictate what data is cached. **Use Proxy for Data Transfer**causes Engines to use the HTTP proxy server

for download of any session's init data. The **Use Proxy for Resource Synchronization**property causes Engines to use the HTTP proxy for resource synchronization download.

If Engines are configured to use a proxy server and the proxy is not available, the Engine does not attempt to download the Service data using alternate connection parameters. It's the administrator's responsibility to make sure the proxy server is up and properly configured. The administrator must consider implementing DNS or IP failover if high availability is required.

Due to the fact that HTTPS requests might not be properly cached in a proxy server, HTTPS is not supported.

Token security for all resource downloads is not supported when using a proxy server. It is assumed that the proxy server is in the same LAN and the LAN is secure. Note that the proxy server cannot download a resource until one of the Engines provides a valid download token.

Note that the following potential problems might occur when using caching proxy servers for resource synchronization:

- Engines can download stale copies of an updated resource from the proxy server. This occurs when the cache timeout is too long or a recently downloaded resource is updated shortly afterward.
- The proxy server might not be able to serve a cached copy of an updated resource if multiple Engines start downloading the resource in a very short time span. This can occur when using Squid even if the `collapsed_forwarding` parameter is enabled.

  To address these issues, adjust the cache size to ensure that the proxy server can cache large files. Additionally, analyzing the resource download pattern and the proxy server configuration is recommended to achieve optimal results. The important factors to consider are:
- Size of the total resources that require downloading
- Size of the total proxy cache, the maximum size per object
- Maximum age of a cached object
- Proxy server behavior for concurrent requests
- Configure the HTTP proxy server to ignore the no-cache header. For example, the following `refresh_pattern` option in the `squid.conf` file causes the squid cache to ignore no-cache headers for URLs matching the regular expression of "`^http://.*/livecluster/resourcesproxy`":

  ```
  refresh_pattern ^http://.*/livecluster/resourcesproxy 0 20% 4320 ignore-no-cache
  ```

Configuring an External Engine Daemon Admin Tool

It is possible to configure an external administration or management tool or system that can be opened from GridServer's Administration Tool. When configured, you can open the tool from the Engine Daemon Admin page with an item in the Actions control. Selecting the new item opens a URL to the external tool. A set of runtime macros enable you to customize the URL.

To configure an external Engine Daemon Admin Tool:

1. In the GridServer Administration Tool, go to **Admin > System Admin > Manager Configuration > Engines and Clients**.
2. In the **Daemon Admin External Tool URL**box, enter the URL to your external tool.
3. The following runtime macros can be used: `${DaemonID}`, `${UserName}`, `${IP}`, `${Status}`, `${PrimaryDirector}`, `${SecondaryDirector}`, and `${OS}`. For example, http://example.com/grid/reports.py?${DaemonID} might be used to look up reports for an Engine Daemon with an external query tool.
4. In the **Daemon Admin External Tool Name**box, enter a description of the configured tool.

To use an external tool, select it from the Actions control next to an Engine Daemon on the Engine Daemon Admin page. It is named "Link to" plus the name you defined in the **Daemon Admin External Tool Name**box.

Quarantine Brokers

In a security-oriented environment, it can be necessary to prevent new or untrusted Engines from joining a grid and downloading potentially sensitive application data or resources until the GridServer administrator explicitly grants permission for them to join. You can exercise this control by using a *Quarantine Broker*, a dedicated Broker in a grid, used only for Engine staging and verification. When Engines do not have permission to log in to other production Brokers on your grid, they can only log on to the Quarantine Broker and await permissioning by an administrator.

The quarantine status is set on any Engine Daemon with the Administration Tool or Admin API. If an Engine Daemon's quarantine status is set to "Verified", the Engines managed by the Daemon might log in to the production Brokers after an Engine Daemon restart. The Engines managed by a quarantined Engine Daemon might only log in to the Quarantine Broker.

Quarantine Broker Concepts

To use a Quarantine Broker, there are two components: the Quarantine Broker, and a method to *Set Quarantine* (send Engine Daemons to the Quarantine Broker) and *Clear Quarantine* (change Engine Daemon status to allow Engine Daemons to log onto production Brokers.)

The Quarantine Broker is specified on the Director. When the Quarantine Broker is specified, the Director allows individual Engine instances to log in to either the production Brokers or the Quarantine Broker based on the value of quarantine status of the managing Engine Daemon. If the quarantine status of an Engine Daemon is “Verified”, all Engines managed by the Engine Daemon might only log in to the production Brokers. Otherwise, the Engines might only log in to the Quarantine Broker.

When using Engine balancing and a Quarantine Broker, the balancing only occurs on the production Brokers. The Quarantine Broker is ignored for the purposes of Engine balancing, and only verified Engines are balanced between the production Brokers.

If the Quarantine Broker is not set in a grid, Engines might log in to any Brokers in the grid allowed by routing and balancing configurations.

Quarantine Status on Engines

The Director determines if an Engine is quarantined by looking for an Engine property called `QuarantineStatus`. The `QuarantineStatus` property value is set to “New Engine” on all newly installed Engines. This ensures that all new Engines are quarantined upon installation when there is a quarantine Broker on the Grid.

When an Engine Daemon has its `QuarantineStatus` property set to any other string than “Verified”, including the null string or having the property missing, it is considered quarantined, and its Engines are only allowed to log in to the Quarantine Broker. When an Engine Daemon has its `QuarantineStatus` property set to the string “Verified”, it is verified, and its Engines are now allowed to log in to other Brokers as per its routing rules.

An unverified Engine Daemon might be cleared from the Administration Tool by changing the `QuarantineStatus` property. This can be automated by using the Admin API. A verified Engine Daemon can be quarantined similarly. There is also an API method that can be called in a Service Session to quarantine an Engine Daemon.

After quarantine status change, the Engine Daemon needs to be restarted for the managed Engines to log in to the intended Broker set. This restriction minimizes the risk of setting the quarantine status by mistake.

Requirements

All Directors require the same Quarantine Broker setting. When failover is configured, this means the Secondary Director has the same Quarantine Broker configuration if the Primary Director fails.

Although a grid might have multiple Brokers, either for redundancy or volume, you can only define one Quarantine Broker in a grid.

When using a Quarantine Broker, you must have an account with **Manager Configure Edit**access in its Security Role (only available in the Configure role by default) to clear quarantine status of an Engine Daemon or set or change the Quarantine Broker definition. An account must have **Engine Properties Edit**access in its Security Role (available in Configure and Manage roles by default) to set Engine Daemon quarantine status. Any role with **Engine Daemon View**access (all roles by default) can see Quarantine Broker and Engine Daemon quarantine status.

Configuring a Quarantine Broker

To add a Quarantine Broker:

1. Install an additional Broker, if you don't already have an extra one installed.
2. Determine the name of the Broker you wish to use. Broker names are automatically given at installation, and are typically numeric. In the GridServer Administration Tool, **Grid Components > Brokers > Broker Admin**shows a list of your Brokers, with their names in the **Broker Name**column. Find the name of the Broker you wish to use for a Quarantine Broker. Note that you can also use this page to change the names of Brokers, if you want to give the Broker a more logical name.
3. Go to **Admin > System Admin > Manager Configuration > Engines and Clients**. Under the heading **Quarantine Broker**, enter the name of the Broker you want to be the Quarantine Broker, then click **Save**. You do not need to restart the Manager.
4. Repeat this configuration (with the same Broker name) for both Primary and Secondary Directory on your grid.

The Quarantine Broker settings persist after Manager restart or future Manager upgrade. If you must change or remove a Quarantine Broker, repeat step 3, but enter a new or blank value for the name.

Setting Quarantine Status on Engines

There are two ways to set and clear quarantine status: interactively with the GridServer Administration Tool, or programmatically with the GridServer Admin API. You can also use the API to self-quarantine Engines. Each method is described below. Also, the following constraints apply when setting quarantine status:

- Due to XML limitations for Engine properties, special XML chars are not allowed in the `QuarantineStatus` property.

- If there is a problem setting the `QuarantineStatus` property, it throws an `AcccessException` (a subclass of `AdminException`) based on the new value.

Using the GridServer Administration Tool

Use the Administration Tool to set or modify the `QuarantineStatus` property in one of the following ways:

- From **Grid Components > E****ngines > Daemon Admin**, select **Edit/View Properties**from the **Actions**list. Select **QuarantineStatus**from the **Properties**column, and enter a new value.
- From **Grid Components > E****ngines > Daemon Admin**, select **Set Property for Daemons on Page**, then select **QuarantineStatus**from the Engine Property List, and change the value of the Property `.`

Using the Admin API

The following method in the `EngineDaemonAdmin` Admin API interface can be used to set and clear Engine Daemon quarantine status:

```
public String setProperty(long id, String key, String value) throws
Exception
```

The value of `key` must be `QuarantineStatus` to set the Engine Daemon quarantine status. The value `id` is the identifier of the interested Engine Daemon. The value might be retrieved by calling `EngineDaemonAdmin.getEngineDaemonIds()`.

The corresponding API call for `setProperty` are also available for C++ (`dsdriver::EngineDaemonAdmin.setProperty`) and .NET (`EngineDaemonAdmin.SetProperty`)

Engine Self-Quarantine Using an API Call

You can also quarantine an Engine using the following method in the Java API:

```
com.datasynapse.gridserver.engine.EngineSession.quarantine (String reason)
throws GridServerException;
```

This method might be called in a Service Session. When this method is called, a synchronous message is sent to the Broker, the Broker then forwards the message plus the Engine session ID to the active Director synchronously. When the Director receives the message, it finds the Daemon ID based on the Engine session ID and sets the Engine property. The Director asynchronously

restarts the Engine Daemon identified by the Daemon ID. If any exception occurs, it propagates back to the Engine. Due to the fact that the Engine ID is not passed from the Engine to the Broker, this API call can not quarantine any other Engine Daemons besides the managing Daemon of the current Engine.

The affected Engine Daemon restarts for the quarantine status change to take effect. When the affected Engines receive the restart command, any running Services are canceled, as usual.

The corresponding API call for `quarantine` are also available for C++ (`dsdriver::EngineSession.quarantine`) and .NET (`EngineSession.quarantine`)

Quarantine Broker Constraints

There is no finer configuration to direct Engine instances to specific Brokers using this feature.

Due to the fact that there is only one quarantine Broker defined in a grid, several issues might arise as a result:

- Quarantined Engines that don’t collocate with the quarantine Broker need to send traffic over the WAN.
- If the Quarantine Broker is not up or accessible, quarantined Engines cannot log in to any Brokers in the grid. This might generate heavy login retry load when the network is partitioned or unstable.
- Drivers are not treated in any special manner with regard to routing to Quarantine Brokers. This is because some might wish to run tests that require a Driver on Quarantined Engines. Routing rules are required to prevent production Drivers from logging in to Quarantine Brokers.

Inside the GridServer Director

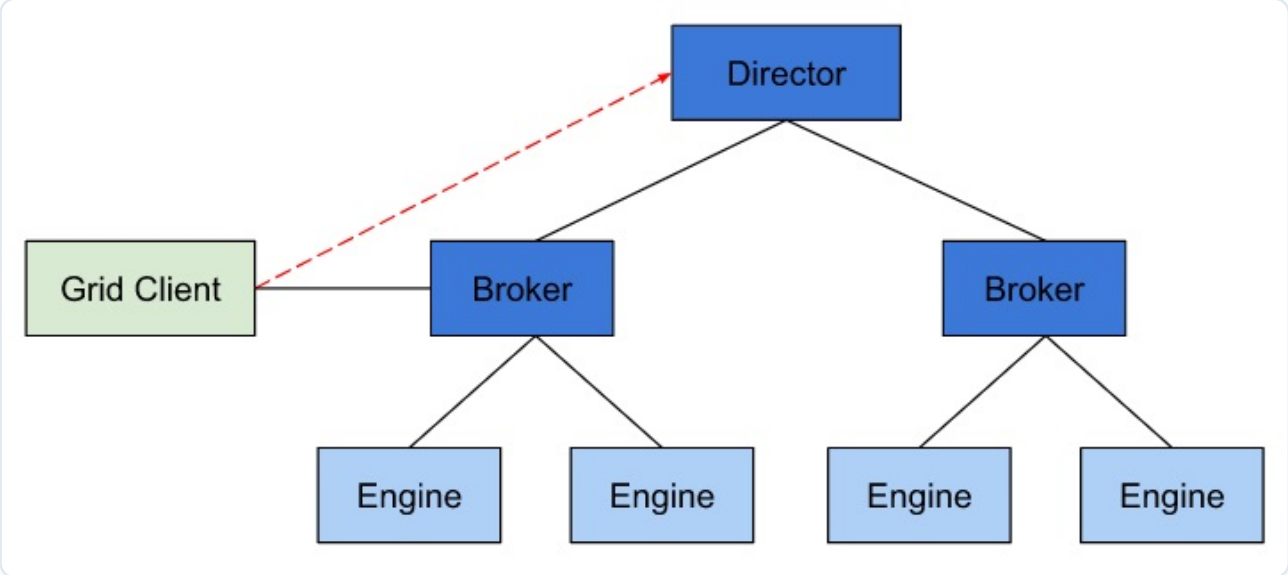
The GridServer Director is part of a GridServer Manager, the controlling mechanism of the Grid. Directors route Engines and Drivers to Brokers, and manage Bro-kers and Engine Daemons. Directors balance the load among their Brokers and provide Broker fault tolerance.

Director Roles

The Director is responsible for three basic roles: routing, authentication, and load balancing.

Routing

One of the GridServer Director's important roles within the Manager is to serve as the routing component. Both Engines and Clients log in via the Director, and the Director routes them to the appropriate Brokers.



Engines and Grid Clients log in to the Director and are authenticated; the Director then routes Engines and Grid Clients to available Brokers.

Authentication

The Director, as part of routing, handles Client authentication.

Load Balancing

Because Directors take a central role with both Client and Engine traffic, they are a key object in configuring the load balancing of components. Directors allocate Clients and Engines to available Brokers.

The default policy for load balancing by Directors is by relative weight; essentially, a Director assigns Clients and Engines to Brokers so that they are equally distributed.

Directors balance load by using the following techniques:

- Clients are routed based on the roles that are assigned to the user running the Client.
- Each Broker can be configured with weight settings for Clients and Engines which balance the number of clients given to that Broker by the Director.
- Idle Engines are relocated to other Brokers if and when needed.

- An Engine configuration can be set to have a Home Broker. A Broker can be set up to have a set of Shared Brokers. Any Engine that uses that configuration is routed only to its Home Broker or any Broker that its Home shares to.

Configuration Management

The Director is used to configure many settings of a GridServer Manager, including users and passwords, Driver profiles, routing properties, and Engine configurations. These settings are configured with the web-based GridServer Administration Tool.

Database Management

GridServer can use an external reporting database to store data such as User, Engine, Driver, and Broker information. The reporting database is an enterprise-grade JDBC database that you provide that is used to log events and statistics. The configuration of the database used is changed on the Director from the GridServer Administration Tool.

Resource Management

Service Deployment files that are used by Engines are centrally managed, starting at the Director. The resources centrally located on the Director are then synchronized to Brokers, which then synchronize them with Engines.

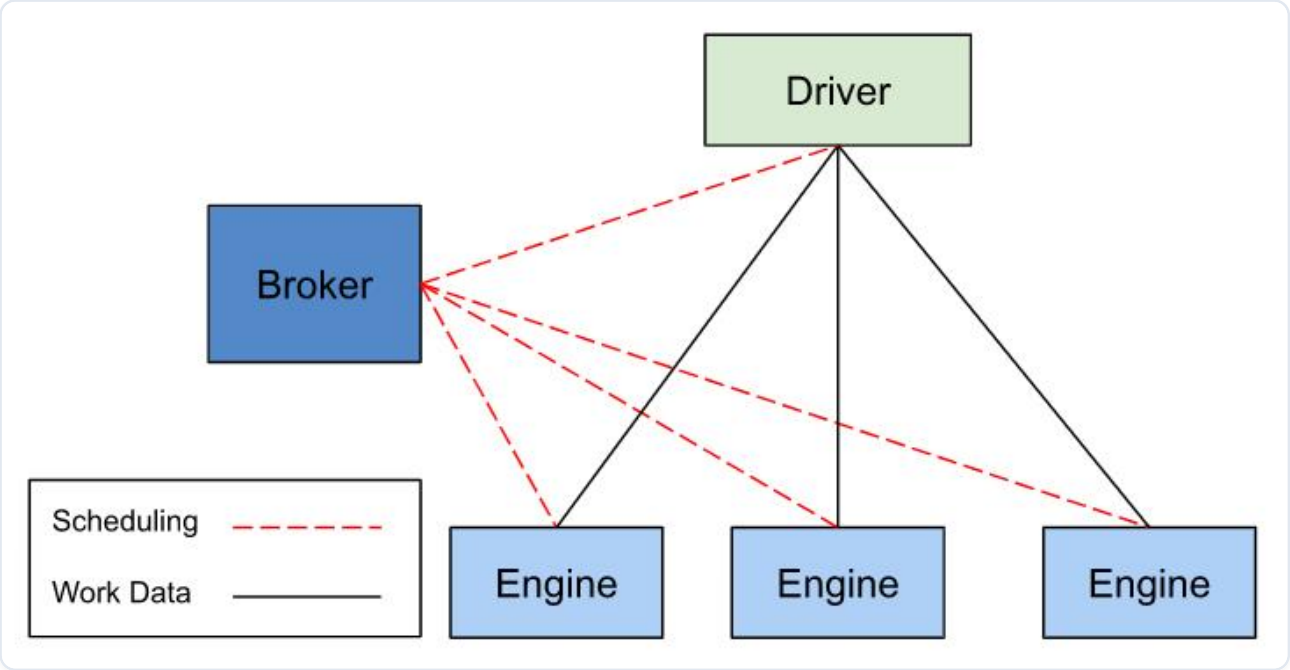
Resources are distributed with *Grid Libraries*. A Grid Library is an archive containing a set of resources and properties necessary to run a Grid Service, along with configuration information that describes how those resources are to be used. Grid Libraries can contain JARs, native libraries, configuration files, environment variables, and hook files or alternate JREs needed to run a Service. Grid Libraries offer the flexibility to do such things as package a Service for use on multiple OSes from a single archive, upgrade resources without interrupting running Sessions, and change Grid Library variables without redeploying a Grid Library.

Inside the Broker

The GridServer Broker, like the Director, is an integral part of a GridServer Manager. Brokers schedule work requests from Drivers to Engines, and manage both types of components to ensure service and return results.

Broker Roles

A Broker’s role includes scheduling, Engine and Driver management, communication, and resource management.



Brokers manage Engines and Drivers, and schedule work with lightweight messages. Drivers and Engines exchange work data directly when using Direct Data Transfer.

Scheduling

Brokers schedule tasks submitted by Drivers onto available Engines. To determine the scheduling of tasks to Engines, Brokers take into consideration Engine state and Service priority.

The Engine state is based on whether that Engine has loaded Grid Libraries needed by that Service, and whether that Session is cached on the Engine. This minimizes the amount of initialization work involved in running a Service on an Engine.

When the Classic Scheduler is used, the priority of a Service is considered when scheduling. Service priority enables more important work to take more resources, and urgent priority work can preempt less important work. The priority of a Service Session is set when created, and can be modified at runtime.

When the SLA scheduler is used, a Session is set to be a member of a predefined SLA group. Such groups are defined on the Broker. The SLA values for groups are either defined as a percentage of Engines on the Broker, or a number of actual Engines. Within a group, Engines are equally distributed to all Sessions in that group. The scheduler guarantees that those SLA values are met, provided there are enough Engines available.

Brokers have several other methods used to control scheduling of tasks:

- Discriminators can be used to select or reject a subset of Engines when scheduling specific tasks or Services. For example, you can use discriminators to target Service requests to Engines running specific operating systems which contain their compiled code.
- Engine Blacklisting provides a method of preventing Engines from taking tasks from a Service if they have previously failed on a task from the same Service.
- The dependency mechanism enables the construction of workflow on a Broker without the need for an active Grid client to manage dependencies (wait for completed tasks, submit more based on successful outcome, and so on). When a Session is submitted, one or more Tasks or entire Services can be required to be completed prior to the Service being scheduled. These dependencies can be Sessions already submitted, or ones that have not yet been created. This way, multiple Sessions in different Services can be submitted, but they are not eligible for scheduling until certain conditions are met—namely the successful completion of specific Sessions.

Brokers handle rescheduling of tasks when Engines go offline. When there is a clean reset of an Engine, such as when its host PC becomes too busy with user activity, it notifies the Broker; the Broker determines if an Engine has failed because it is monitoring its connection. In either case, any outstanding work is rescheduled to other Engines.

Engine and Driver Management

Although authentication and assignment of Engines and Drivers is handled by the Director, the Broker manages clients through the remainder of their life cycle. This involves determining the availability of clients by monitoring their connections, and coordinating these results with the scheduling effort undertaken by the Broker.

Communication

The GridServer architecture combines the central control of a hub and spoke architecture with the efficient communication of a peer-to-peer architecture. This is made possible by the flexibility in configuring communication between the Broker, Engines, and Drivers.

Engines and Drivers communicate with Brokers using lightweight transaction messages. Direct Data Transfer (DDT) enables Engines and Drivers to exchange work data directly. This peer-to-peer communication eliminates any “heavy lifting” between the Broker and clients, improving its performance.

DDT can be disabled in situations where network layout requires all communication to go through a central point like a Broker. This causes both transaction messages and data to be transferred

through the Broker, creating a more traditional hub and spoke architecture between Brokers and clients.

Resource Management

Brokers maintain a copy of deployed resources from the Primary Director, and Engines synchronize these resources from the Broker.

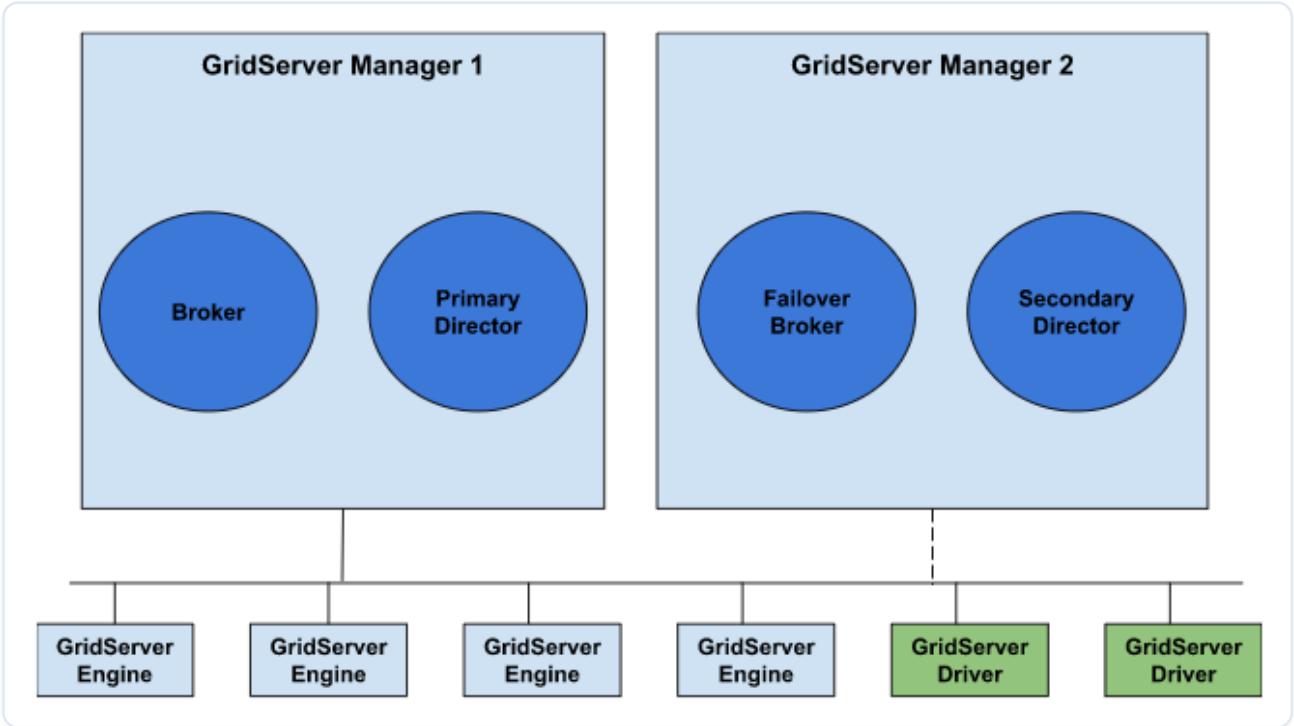
Inside GridServer Topology

The GridServer *Manager* loosely consists of two entities: the GridServer Director and the GridServer Broker:

A minimal configuration of DataSynapse GridServer consists of a single Manager configured with a Primary Director and a single Broker. Additional Brokers or Directors can be added to address three primary concerns: redundancy, volume, and application segregation.

Redundancy

Given a minimal configuration of a single Director and single Broker, Engines, and Drivers log in to the Director, but failure of the Director (such as in the case of hardware or network failure) means a Driver or Engine not logged in, is no longer able to establish a connection.



To prevent this, redundancy can be built into the GridServer architecture. One method is to run a second Manager with a Secondary Director, and configure Engines and Drivers with the address of both Directors. If the Primary Director fails, Engines and Drivers contact the Secondary Director,

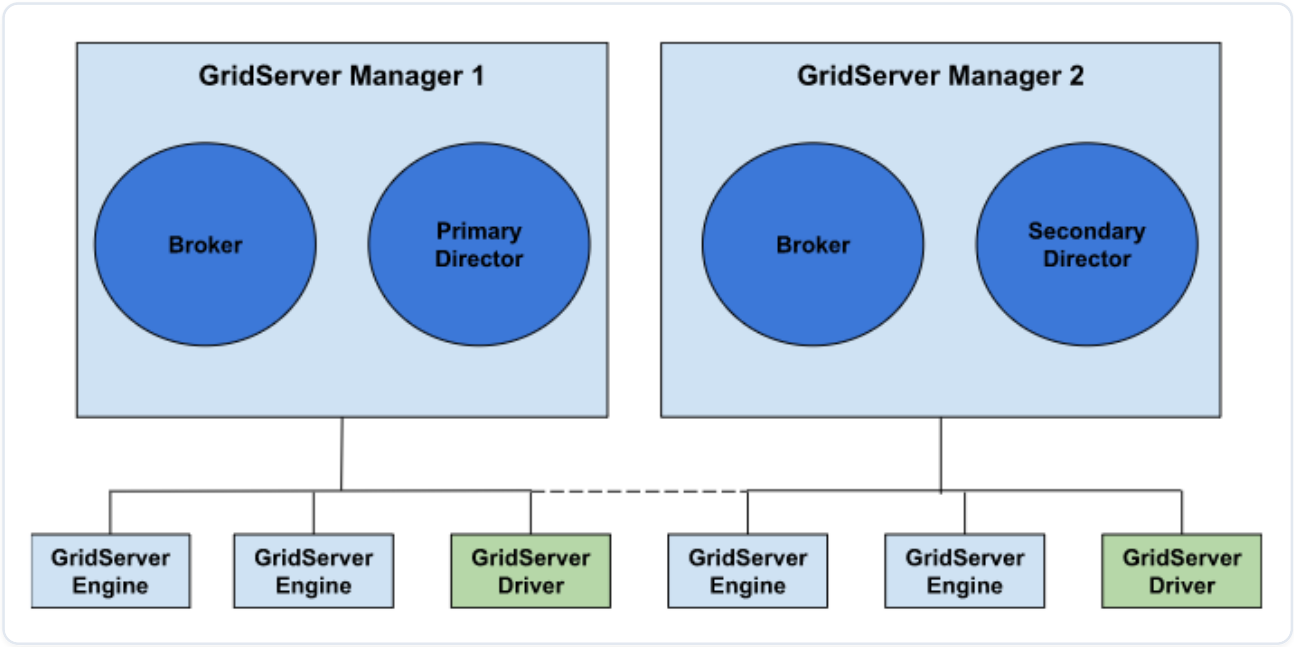
which contains identical Engine configurations and route Engines and Drivers to Brokers in the same manner as the Primary Director. The figure above shows an implementation with two Managers.

In addition to redundant Directors, a Broker can have a backup on another Manager. A Broker can be designated a Failover Broker on a Manager during installation or in the Manager Configuration page. A Failover Broker acts as a backup Broker for times when other Brokers are offline. Under normal operation, they are idle.

For more information about redundancy, see the TIBCO GridServer® Administration.

Volume

In larger Grids, the volume of Engines in the Grid might require more capability than offered by a single Broker. To distribute load, additional Brokers are added to other Managers at installation. For example, the figure below shows a two Manager system with two Brokers



Topography

Several other factors might influence how you integrate DataSynapse GridServer with your computing environment. These include:

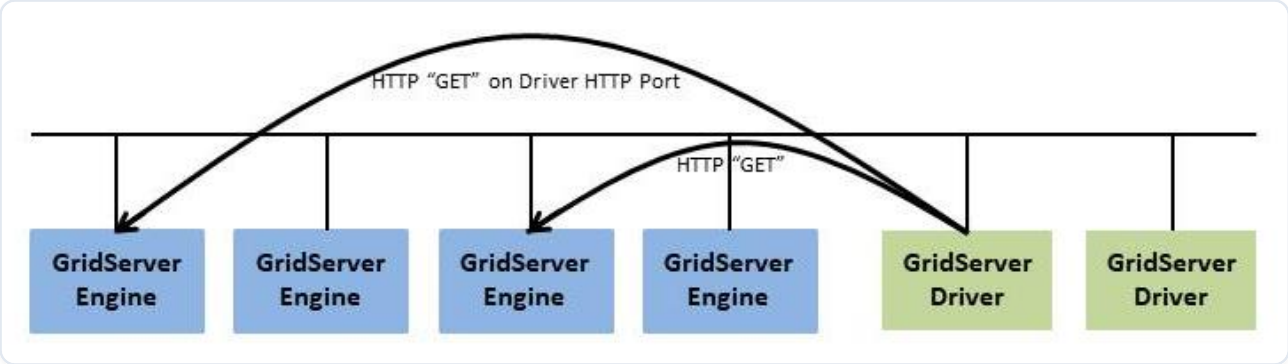
- Instead of using one Grid for all types of Services, you might wish to divide different subsets of Services (for example, by size or priority) to different Brokers.
- Your network might dictate how your Manager environment must be planned. For example, if you have offices in two parts of the country and a relatively slow extranet but a fast intranet in each location, you could install a Manager in each location.

- Different Managers can support data used for different types of Services. For example, one Manager can be used for Services accessing a SQL database, and a different Manager can be used for Services that don't access the database.

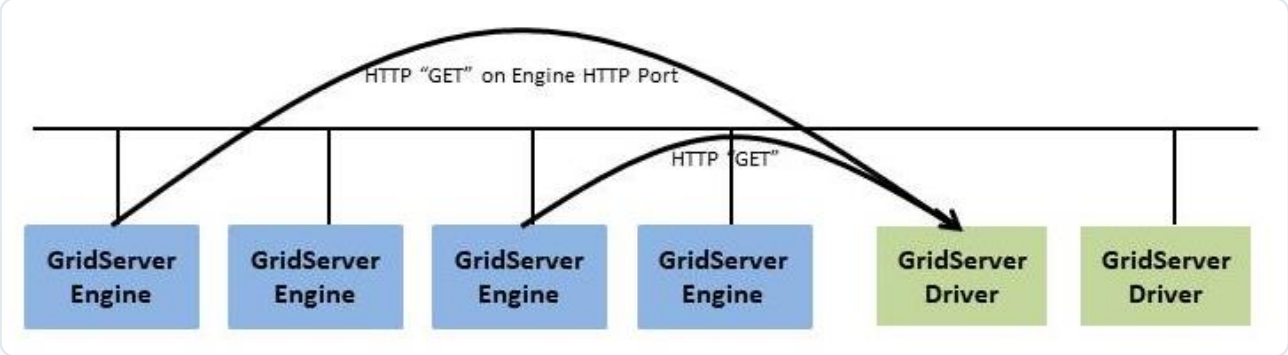
With this flexibility, it's possible to architect a Manager model to provide a work space that facilitates your workload and traffic. For more information about how to best design your Manager environment, contact our Integration Services staff, and we can help you determine how to best configure your installation.

Direct Data Transfer

By default, GridServer uses Direct Data Transfer, or peer-to-peer communication, to optimize data throughput between Drivers and Engines. Without Direct Data Transfer, all task inputs and outputs must be sent through the Manager. Sending the inputs and outputs through the Manager results in higher memory and disk use on the Manager, and might lower throughput overall.



With Direct Data Transfer, only lightweight messages are sent through the Manager, and the "heavy lifting" is done by the Driver and Engine nodes themselves.

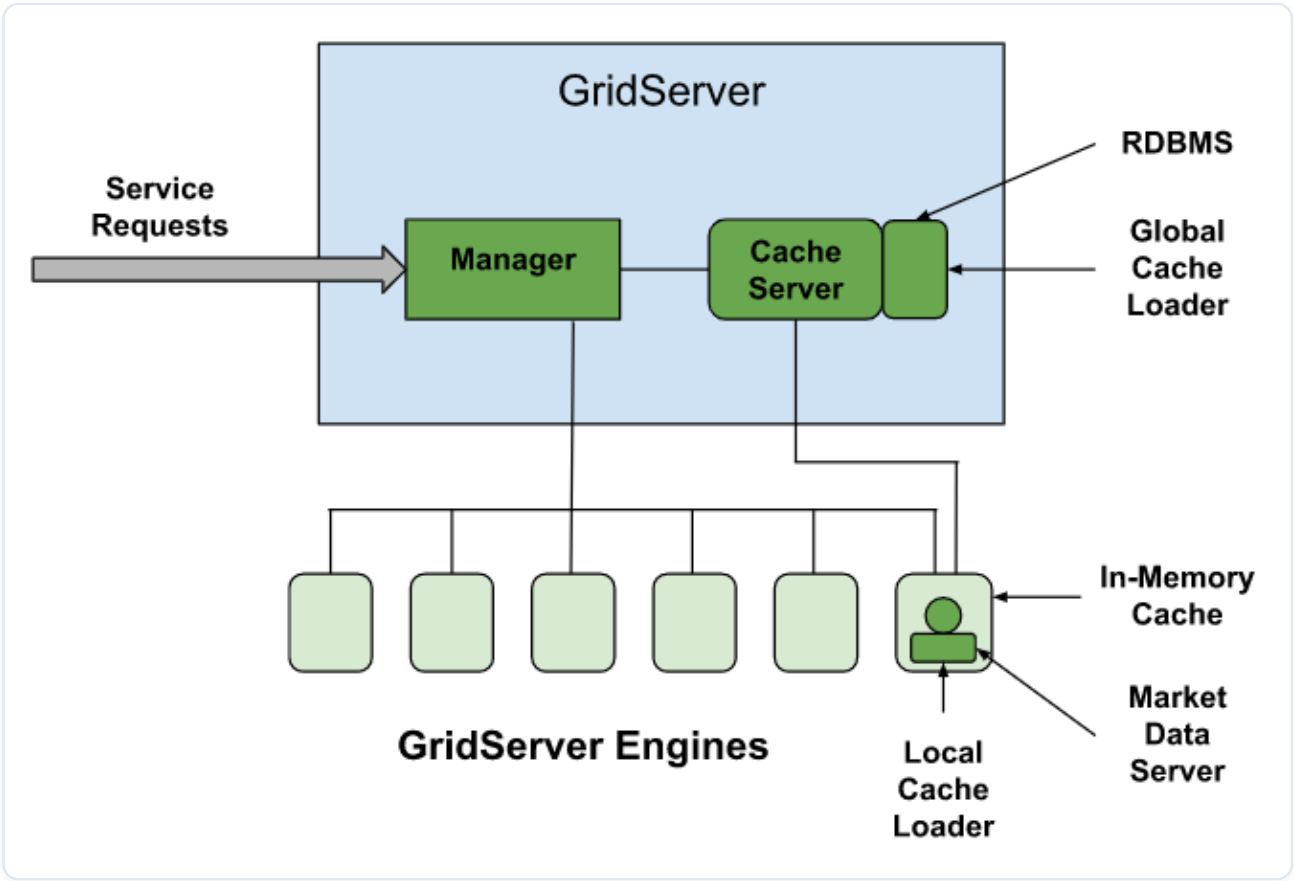


GridCache

This section provides information about using GridCache, a dynamically updateable distributed object cache that any GridServer Driver or Engine can use to store data for later retrieval by other GridServer components.

Overview

You can dynamically update *GridCache*. It is a distributed object cache that any GridServer Driver or Engine can use to store data for later retrieval by other GridServer components. While GridServer extensively uses object caches internally, applications can access the GridCache object cache directly through an API, to reduce load on backend datastores and decrease access time to data. GridCache is similar to the proposed JSR-107 JCache, with a similar interface and a subset of features.



GridCache meets the requirements of many informational market data systems, where a consistent view of object state is extremely important but it is not necessary to guarantee that all participants process every individual state change (for instance every individual quote move) as a transaction.

General Capabilities

GridCache is a general distributed cache that provides a consistent view of data to all clients (Drivers and Engines) in the Grid. Data resides in unique *regions* of the cache. Data can be serializable Java objects, .NET objects, strings, and byte arrays for C++. The global cache of data can be arbitrarily large, limited only by the amount of disk space on the Manager. Each component locally caches only the data requested by users on that component. The local cache of each client, designed to speed up access to frequently used data, is in-memory with the option in Java Drivers

and Engines to spool to disk. The size of the local cache is configurable through the Engine distribution configuration or the Driver properties file.

API

You can access GridCache through a client API available on Drivers and Engines. The API follows the JCache specification where appropriate. The API is available in Java, .NET, and C++, and provides cross-platform access to data where appropriate. That is, C++ and Java applications can share XML documents (strings), but have little use for sharing Java or .NET objects. You can also use Data References across different platforms to support streaming very large objects. For more information, see "Data References" in TIBCO GridServer® Developer's Guide.

Modes

GridCache operates in one of the following modes:

- **Local** This mode lets you cache data locally by putting elements into the cache. This mode does not synchronize clients that are accessing local cache regions with the same name. This is similar to having a local hashtable with LRU and eviction based on time-to live from creation time.
- **Local with loader** This mode lets you load data into the local cache using a loader specified at create time. Puts (cache writes) are not allowed in this mode. Users can manually synchronize clients' local caches using clear and invalidate methods.
- **Global** Data that users put into the cache is then available globally. Full automatic synchronization occurs in this mode. All components have access to a synchronized view of all entries.
- **Global with loader** Users can use a global loader to load data into the cache. Full automatic synchronization occurs in this mode.

Cache Configuration and Access

To configure cache regions, use the GridServer Administration Tool at **Services > Services > Cache Regions**. Define the region names or regular expressions with a set of attributes. The `getRegion` method in `CacheFactory` provides access to the region if it already exists or creates the region with the mapped attributes. A region name throws an exception when it matches multiple regular expressions from different schemas. A second Cache access method is available and takes the schema name to provide dynamic mapping of regions to attribute schemas.

When you configure Local and Global loaders, the class name of the loader and the type are arguments to the constructor. Users can define bean properties for loaders. Loaders are available

only in Java and .NET, but can be used by the C++ API. JNI, or managed C++ can be used to implement loaders to access native resources. Each schema that requires a loader defines the loader within the configuration page.

Changes to the cache configuration take effect the next time you create regions with that cache configuration. Pre-existing regions that require the configuration changes must be manually destroyed and recreated.

Data Storage

All data put to a global cache is stored in a persistent backend datastore on the Broker's file system. You can limit the size of the Broker's file cache. The default is that there is no limit. If the Broker's file cache is size limited and full, the Broker silently removes data to make room for the incoming data. If the cache entry is too big, it first goes to the memory cache, and if it cannot fit there it goes to the disk cache. If it is too big for the disk cache too, then it is dropped. There is no global resilience when using a memory-based cache; however, you can use a loader for that purpose.

Attributes

Define the GridCache Configuration Attributes in schemas. Then, apply the attributes to newly created regions. You can define the following types of attributes:

- **TimeToLive**Regions can define a time-to-live attribute. Data that is stored in the cache for longer than the time-to-live attribute is evicted, and you must reload that data from the backend datastore. Data in local caches is evicted locally. Data for distributed regions is evicted from the distributed cache, and the Broker and all clients delete that data if they have cached it locally.
- **Local/GlobalLoader**Loads data from a backend datastore. See Cache Loaders, below.
- **KeepAlive**Specifies how long the client keeps the region and its keys in its local cache after the last reference to the region goes away.

Consistency/Synchronization

The cache synchronizes by propagating update notifications to all clients listening on a region. These notifications occur any time the region changes. Specifically, they occur on a put (when an element already exists), clear, remove, or invalidate. This applies to different region types differently:

- **Engines**GridCache guarantees that all Engines receive all update notifications by the time they take the next task or Service request.

- **Drivers**There are no synchronization guarantees for the Driver. The Driver receives notification messages the next time it performs a request, polls the server for results, or sends a heartbeat.

Cache Loaders

Loaders provide an optional mechanism for loading data into the cache from a backend datastore, such as a relational database. Users can implement and associate Cache Loaders with a region of the cache. These Cache Loaders can be installed locally on the client (Driver or Engine) or globally, in the GridServer Broker.

Global

Use a Global Cache Loader for synchronized regions from which all clients can access data.

- Global loaders are defined and configured in the schemas.
- When other clients get access to that region, they automatically are using that loader.
- A client can specifically pre-load data into the cache by explicitly calling the load method with a single key or a list of keys.
- If a get does not find data, the loader then attempts to load for that key by calling the loader's load method.
- Puts are not allowed on regions with loaders.
- Global loaders are written in Java, but can be bridged to native or .NET code through JNI.
- Global CacheLoader JAR files are deployed to the `lib` directory in the cache directory. By default, the cache directory is `DS_DATA/cache`. Configure the cache directory in the **Cache** section of the **Manager Configuration**page in the Administration Tool.

Local

Use a local loader to cache data locally from an external database. Clients do not share local loaders.

- Puts (writes) are not allowed, as it is a local cache, and data is not propagated to other clients or regions.
- Removing an item is not allowed. Instead, invalidate the item. Invalidating causes the item to be removed from other client's caches.
- Local loaders can only be .NET or Java. You can adapt CPP loaders through JNI or managed C++.

Cache Loader Write-through and Bulk Operations

To simplify using back end datastores with GridCache, it's sometimes desirable to add a means for supporting store and remove methods on the loader rather than just supporting it purely as a data fetching mechanism. It's also sometimes desirable to be able to use such methods in a batch form, such that a cache can be "primed" with entries through a bulk load method, or that multiple objects can be stored, removed and invalidated with a single method to cut down on per-store operation overhead.

Loaders can inherit from two interfaces that support methods for supporting store and remove methods on the loader:

- The preload methods are exposed in the `BulkCacheLoader` interface.
- The store, remove and clear methods are exposed in another interface, `CacheStore`, and can be used by the underlying cache mechanism for modifying the content of cache puts and removals on the backend datastore. The cache mechanism can then invalidate the corresponding entry or entries for all other caches listening on that region, if applicable.

It is possible that the backend store gets updated without sending an invalidation message to all other clients. If this scenario is detected, it throws an exception indicating a loss of synchronization, but the cache client must handle recovery from that point on.

The caching system in GridServer does not provide a mechanism to auto-update data in the cache when it changes in the backend, if done so by a mechanism other than those offered by the CacheStore interface.

Support is available for datastore write-through, bulk write-through, remove, bulk remove and bulk load, on both global and local loaders, in Java and .NET.

Notification

GridCache provides an optional mechanism whereby you can implement a class that listens for update notifications. An update is defined to be either an invalidation call on a loaded object or on a put call on a key that exists in the cache already. You can then take any action desired such as updating local copies of the object or data to the new version or ignoring the update completely. The next time that the data is requested from the cache, GridCache fetches and locally caches the most current version of that data.

Disk/Memory Caching

When the cache is full, cache puts (or writes) push the oldest element out of the cache. Elements are then put into the backing disk cache or removed entirely. This makes the caches, LRU caches. You can configure the size of the local cache and the size of the backing disk cache:

- In the `driver.properties` file, for Drivers.
- In the Engine configuration, for Engines.

If you configure disk caching, then any puts into the memory cache when the memory cache is full force the oldest element out of the memory cache into the disk cache. Any access to a cache element that has to get the element from the disk cache brings the element into the memory cache. CPP Drivers do not have disk-backed cache.

Cache Region Scope

Global cache regions exist until they are destroyed through the destroy method regardless of whether any client has a reference to that region. Unnecessary global cache regions impact eviction performance, so it is important to destroy global regions when they are no longer needed.

After all references to a cache region on a client go out of scope, local cache regions persist on clients until their keepalive timeout. At that point, the region is swept from the cache. Use the `close()` method to explicitly release a reference to a region. If you do not use the close method, garbage collection handles decrementing references to the region. However, garbage collection is never guaranteed so the keepalive timeout is not a guaranteed timeout. Using the close method is recommended.

Data Conversion Matrix

The following table outlines the results of putting a data type into GridCache and then getting it with a different Driver.

INPUT	.NET OUTPUT	JAVA OUTPUT	C++ OUTPUT
String	String	String	std::string
Java or .NET byte	byte	byte	std::string
.NET Object	Object	undefined	undefined
Java Object	undefined	Object	undefined
DataReference	DataReference	DataReference	DataReference

Using The GridCache API

Details about the GridCache API are in the GridServer API JavaDoc, which is available from the Documentation page of the Administration tool. Documentation for the `Cache` interface covers the use of GridCache.

The GridCache API supports the following seven primitives:

GridCache constructor with CacheFactory

To create a new GridCache instance, use the CacheFactory to get a reference to a particular region. On a particular client component, you can construct multiple instances of a GridCache with the same region, but each exposed instance with the same region shares the same underlying implementation. This lets multiple Sessions share the same view of a cache without having to duplicate the storage or the code.

Put and Get

The `put` method writes to the cache a new entry for a key and object. The `get` method returns the object stored in the cache, for a given key. If you use the get method on a key that does not exist and the region has an associated loader, the loader attempts to load the data for that key. A second get method, when given a mapping of keys, bulk-gets a mapping of keys and their objects. Bulk gets use a single HTTP request for each map, which can lower transaction overhead.

Region

Region names are any printable low ASCII character (32-126), except for characters prohibited in XML attribute values (', ", &, <).

Keys

A Key is a string that refers to an object in the cache. The `keys` method gets:

- For a global region type, a list of all keys currently stored on the Manager for this cache.
- For a local region type, a list of locally cached keys.

Remove

Removes this object from the region, from the Manager, and from all distributed caches.

Clear

Clears all objects from the region, the Manager, and regions on other components.

Invalidation handlers

By default, GridCache implements a lazy invalidation mechanism where callers are told only that their version of an object is out-of-date when they make a fresh “get” call for the object. The invalidation handler interface lets the caller register or deregister to receive asynchronous notification that a get, put, remove, or clear has invalidated the caller’s local copy of an object.

Note that .NET cache loaders for GridCache must be packaged as a Super Grid Library. For more information about using Super Grid Libraries, see the TIBCO GridServer® Administration.

Fault Tolerance and GridCache

GridCache supports fault-tolerance. For details, see the TIBCO GridServer® Administration.

Inside the Client

Grid clients connect and submit requests to GridServer through a GridServer *Driver*. There are many different options and the intent of the flexibility is to allow rapid integration with existing systems.

The GridServer Drivers provide simple API access to Grid Services. The invocations of these Services are virtualized, meaning requests are submitted to a queue, and the Service is eventually performed by a GridServer Engine. Many requests can be submitted simultaneously with GridServer performing the requests in parallel.

Client Types

Several types of Client interfaces are available for use with different applications.

CLIENT INTERFACE	DESCRIPTION
JDriver	The Java Driver, also known as JDriver, consists of a JAR file used with your Java application code.
CPPDriver	The C++ Driver consists of DLLs and include files that are linked with your GridServer application.
.NETDriver	The .NET Driver consists of a DLL that includes a .NET Assembly used with your .NET

CLIENT INTERFACE	DESCRIPTION
	code. The .NET Driver is only available for Windows.
COMDriver	The COM Driver enables an application using the Windows Component Object Model architecture to access GridServer. The COM Driver is only available for Windows.
R Driver	The R Driver enables R applications to access Services. The R Driver is available for Windows and Linux.
Python Driver	The Python Driver, or PyDriver, is a Driver that can submit Python scripts for execution in the GridServer environment. This enables you to create Python scripts to run on multiple Engines, and return the results to a central location. You can achieve parallelism by iteratively changing the script that is passed to successive tasks.
PDriver	*PDriver*, or the Parametric Job Driver, is a Driver that executes command-line programs as a parallel processing service using the GridServer environment. This enables you to take a single program, run it on several Engines, and return the results to a central location, without writing a program in Java or C++.

Installation and Configuration

Drivers are distributed in the GridServer SDK The software development kit includes Drivers, API documentation, and code examples. The SDK is downloaded from the GridServer Administration Tool.

Drivers use a local file called `driver.properties` to define properties used by the Driver at startup, such as the names of the Directors it uses, and what log files are generated. You must also specify the Driver's username and password, and must be a valid GridServer user. (Kerberos can

also be used instead of including a password.) The user’s roles might limit what Brokers the Driver is allowed to be routed to.

For more information about Driver Installation, see the *TIBCO GridServer® Developer's Guide*.

Inside the GridServer Batch Processing Facility

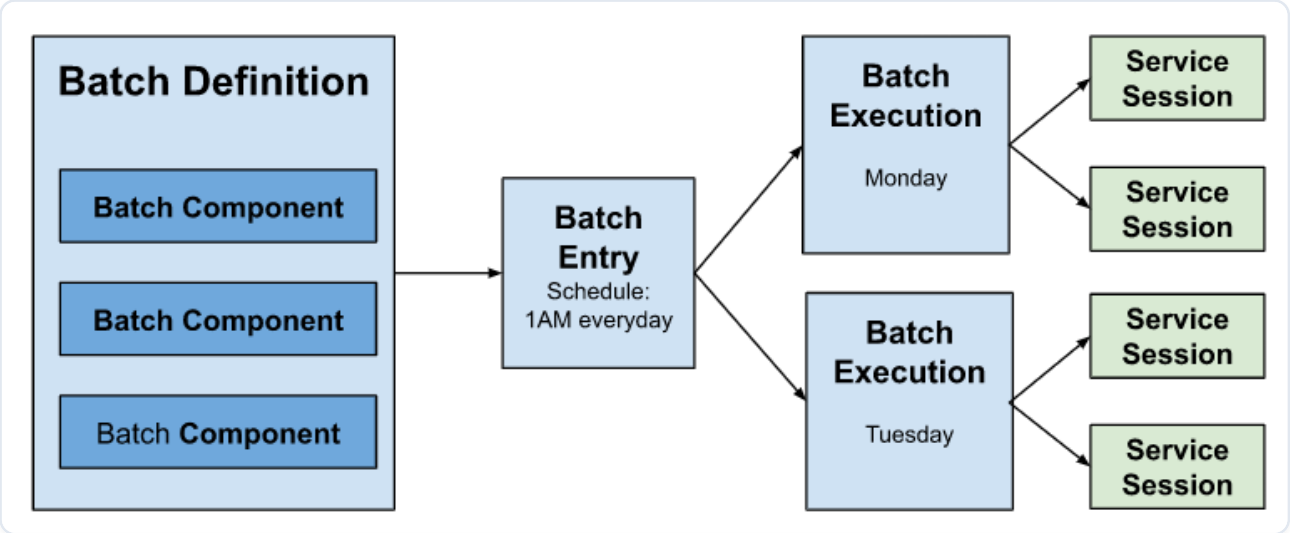
GridServer comes with a built-in batch scheduling facility that makes it easy to schedule Grid workflows from a Web browser. Services can be scheduled from this facility, as well as Grid administrative functions. For example, a user could schedule all the Engines to reconfigure each night at a specific time. Another example is to re-assign Engines to Brokers throughout the day. The most common use, however, is scheduling fixed Grid workloads at specific times throughout the day or week.

Batch events and commands are extensible through a Java API. A programmer can create arbitrary events that trigger processing, such as a database event or messaging event from an external system. GridServer comes with a FileEvent that watches a file to see if it's been modified. Once modified, it continues processing the batch plan.

An interface is defined for commands that allow the developer to perform any action in Java. This action can then be included in a defined workflow. GridServer comes with commands for emailing notification, as well as running services defined in the Service Runner Registry.

Parts of a Batch

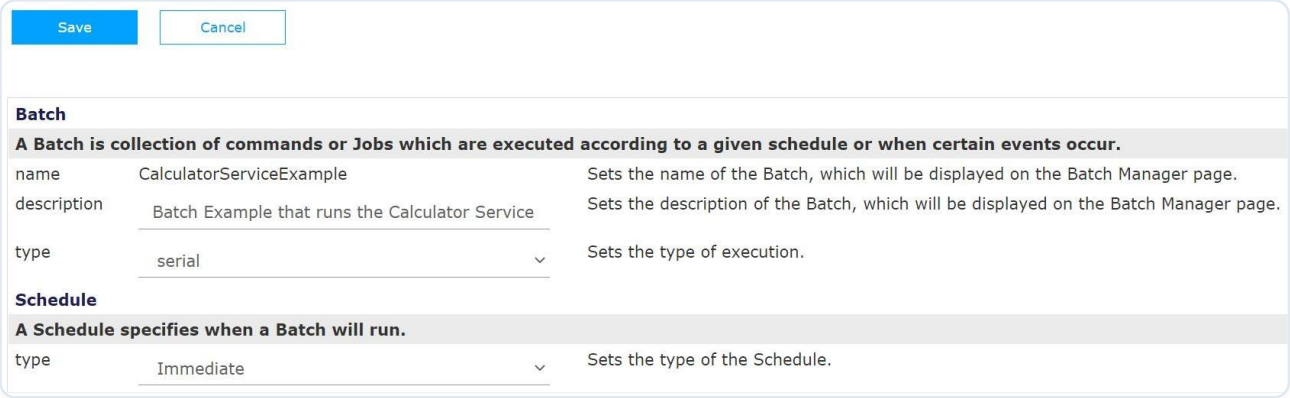
A *Batch Definition* contains instructions in the form of components that define both the schedule and the components that are executed. When the Batch Definition is scheduled on the Manager, it creates a *Batch Entry*, which typically waits until its scheduled time, then creates a *Batch Execution*. The Batch Execution then creates Service Sessions, PDriver Jobs, or other Grid workloads that are executed on the Manager.



A Batch Definition contains Batch Components; each of which defines what the Batch Definition does. When a Batch Definition is scheduled, it creates a Batch Entry, which is an instantiated Batch Definition and runs as defined by the Batch Components. When it runs, it creates a Batch Execution, which then runs the components according to the definition.

Batch Definitions

Batch Definitions are created using the GridServer Administration Tool, with a GUI-based Batch Editor. The Batch Editor contains a list of Batch Definitions resident on the Manager, and enables you to edit, delete, rename, or schedule any of these Batch Definitions. When you select a Batch Definition to be edited, a window displaying all of the Batch Components, their properties, and the order in which they are executed is opened, as shown.



The Batch Editor

Batch Components

By default, a Batch Definition contains a Batch component, and that contains a Schedule component. Each component contains parameters, and values are entered in text boxes or are changed with list controls. The editor provides built-in help to explain components, parameters, and their function.

The Batch Editor page enables you to change the order that Batch components are executed, and add or remove components from a Batch Definition.

Batch components on the Batch Editor page can define when Batches start, run Services, change Engine weights, send email messages, wait for an event to occur, test conditionals, and much more.

Scheduling Batch Definitions

The Batch Registry page lists all of the Batch Definitions on the Manager. You can schedule the Batch Definition, create a Batch Entry, and open the Batch Schedule page.

Administering Batches

Batch Entries on a Manager are listed and administered on the Batch Admin page. All Batch Entries resident on the Manager are listed. You can suspend, resume, remove, or edit an existing Batch Entry.

For more information about the Batch Facility, see the *TIBCO GridServer® Administration*.

Administration Guide

- Typographical Conventions
- Introduction
  - Before You Begin
- The GridServer Administration Tool
  - Overview
  - Getting Started
  - Navigating the Administration Tool
- User Accounts and Role-Based Access Control
- Managing Users
  - About Authentication and Authorization
    - Types of Users
    - Role-Based Access Control
  - Using GridServer Built-In Authentication
  - Using GridServer Built-In Authorization
    - Creating User Accounts
    - Resetting User Accounts From the Command Line
  - Using LDAP Authentication and Authorization
    - Configuring GridServer for LDAP Authentication or Authorization
    - Authentication Schemes Supported in Comparison Mode
    - Security Notes
    - LDAP Configuration Example
  - Using Windows Authentication and Authorization
    - Configuring Windows Authentication
    - Configuring Windows Authorization

    - Using Pure Kerberos Authentication
        - Configuring Pure Kerberos Authentication
        - Configuring Pure Kerberos Authorization
    - Managing Multiple Brokers with Grid Single Sign-On (SSO)
        - Grid SSO Configuration
        - Constraints and Limitations
    - Client Routing
        - Routing Clients With Roles
        - Routing Clients On The Broker Routing Page
        - Routing Clients With The Driver API
- Managing Services
    - Deploying Services
        - About Grid Libraries
        - Using Grid Libraries from a Service
        - Super Grid Libraries
        - Deployment
        - Disabling Resource Deployment
        - Bridges
        - JREs
        - Packaging Grid Libraries
        - Distributing Grid Libraries
        - Grid Library Filters
        - JAR Ordering File
        - Uploading and Deploying with the Admin API
    - Running Services
        - Registering a Service Type
        - Service Run-As

        - compressData
        - encryptionEnabled
        - Using Run-As
    - Scheduling
        - Reschedules and Retries
        - The Scheduler
        - Common Scheduler Features
- Managing Engines
    - Engine Routing and Balancing
        - Balancing and Service Discriminators
        - Engine Weight-Based Balancer
        - Home/Shared Balancer
        - Engine Balancer Configuration
        - Engine Upper and Lower Bounds
    - Failover Brokers
    - Example Use Cases
        - N+1 Failover with Weighting
        - Engine Localization with Sharing
    - Engine Configuration
        - Editing an Engine Configuration
        - Creating a New Engine Configuration
        - Copying an Engine Configuration
        - Setting the Engine Configuration Used by Engines
        - Setting the Director Used by Engines
        - Configuring Engines With Multiple Network Adapters
        - Configuring Engine Daemons to Use SNAT
        - Using the System Classloader on an Engine

    - Configuring a Global Shared Grid Library Directory
    - Configuring When Engines Run
        - Manual Mode
        - Auto Mode
    - Configuring How Many Engines Run
    - Running Engines in Multiplexed Mode
        - Communication and Task Scheduling
        - Configuration
    - Configuring 64-bit Engine Daemons to run 32-bit Services
        - Configuration
        - Specifying that a Service is win32
        - Routing 32-bit Tasks to 64-bit Engines
    - Configuring a Caching HTTP Proxy Server
    - Configuring an External Engine Daemon Admin Tool
    - Quarantine Brokers
        - Quarantine Broker Concepts
        - Quarantine Status on Engines
        - Requirements
        - Configuring a Quarantine Broker
        - Setting Quarantine Status on Engines
        - Quarantine Broker Constraints
- Grid Fault-Tolerance and Failover
    - The Fault-tolerant GridServer Deployment
    - Heartbeats and Failure Detection
    - Manager Stability Features
    - Engine Failure
    - Driver Timeout and Failure
    - Director Failure

    - Broker Failure
    - Failover Brokers
    - Task Fault Tolerance
    - Batch Fault-Tolerance
    - GridCache and PDriver Fault-Tolerance
- Administration and Maintenance
    - Configuration Issues
        - Installation on Machines With Multiple Network Adapters
        - Using UNC Paths in a driver.properties File
        - Renaming a Broker
        - Moving a Manager
    - GridServer Manager Administration Procedures
        - Backup / Restore
        - Importing and Exporting Manager Configuration
        - Setting the SMTP Host
        - Configuring the Timeout Period for the Administration Tool
        - Reconfiguring Managers when Installing a Secondary Director
        - Reconfiguring the Engine Communication Port
        - Promoting a Secondary Director to Primary Director
        - Configuring SNMP
    - LogLogic Integration
        - Configuration
        - Logging Message Format
        - Output
    - Elasticsearch Integration
        - Configuration
        - Message Format

    - Database Maintenance
        - Database Types
        - Internal Database Reset
        - Internal Database Backup
        - Performing Reporting Database Maintenance
    - The Batch Scheduling Facility
        - Terminology
        - Editing Batch Definitions
        - Batch Components
        - Service Runners
        - Scheduling Batch Definitions
        - The Batch Schedule Page
        - Running Batches
        - Deploying Batch Resources
        - Batch Fault-Tolerance
- Optimizing the Grid
    - Diagnosing Performance Issues
    - Tuning Data Movement
- Diagnosing GridServer Issues
    - Troubleshooting Overview
    - Reporting an Issue
    - Obtaining Log Files
        - Manager Logs
        - Engine and Daemon Logs
        - Application Server Logs
        - Monitoring the Tomcat Application Server
        - Monitoring Engines Using JMX
    - Diagnosing Network Issues

    - Diagnosing Engine Issues
    - Diagnosing Driver Issues
    - Diagnosing Manager Issues
        - Manager Port Issues
        - Out of Memory Issues
        - Deployment Issues
        - GridCache Issues
        - Database Issues
    - Troubleshooting Tools
        - Task Admin Page
        - Task Queue Dump
        - Enabling Enhanced Task Instrumentation
        - Process Explorer
        - Dependency Walker
- Event Streaming by Using Apache Kafka
- Reporting Database Tables
    - Data Type Mapping
    - batches
    - brokers
    - broker_stats
    - driver_events
    - engine_events
    - engine_info
    - engine_stats
    - event_codes
    - jobs
    - job_status_codes
    - roles
    - tasks

    - task_status_codes
    - user_events
    - users
- Scheduler Instrumentation Database Table
    - scheduler_info
- Engine Instrumentation Database Table
    - engine_ins

Typographical Conventions

The following table lists the typographical conventions used in this guide:

CONVENTION	USE
`TIBCO_HOME`	Many TIBCO products must be installed within the same home directory. This directory is referenced in the documentation as `TIBCO_HOME`. The default value of `TIBCO_HOME` depends on the operating system. For example, on Windows systems, the default value is `C:\tibco`.
`DS_INSTALL`	TIBCO GridServer® installs into a directory within `TIBCO_HOME` named datasynapse. This directory is referenced in the documentation as `DS_INSTALL`. The default value of `DS_INSTALL` depends on the operating system. For example, on Windows systems, the default installation directory is `C:\tibco\datasynapse`.
`DS_MANAGER`	The *Manager directory* contains the read-only software files; by default, it is a directory within `DS_INSTALL` named `manager`, and is referred to as `DS_MANAGER`. For example, on Windows systems, the default Manager directory is `C:\tibco\datasynapse\manager`.

CONVENTION	USE
`DS_DATA`	The *data directory* is the location of all volatile files used by the application server such as server properties and configuration. By default, it is a directory within `DS_INSTALL` named `manager-data`, and is referred to as `DS_DATA`. For example, on Windows systems, the default data directory is `C:\tibco\datasynapse\manager-data`.

Introduction

The *TIBCO GridServer® Administration* is for administrators who maintain DataSynapse GridServer installations. It describes how GridServer works and how to use the GridServer Administration Tool. Topics include how to schedule and route Services, deploy resources, manage failover Brokers, and perform other frequent tasks. This guide also provides advanced information about security, tuning, database administration, and log files.

Before You Begin

This guide assumes that you know GridServer concepts. If you do not, see the TIBCO GridServer® Installation *Guide* for information about the GridServer component architecture and principles of operation.

Before beginning, you must already have a DataSynapse GridServer Manager running and know the hostname, username, and password. If this isn't true, see the TIBCO GridServer® Installation*Guide* or contact your administrator.

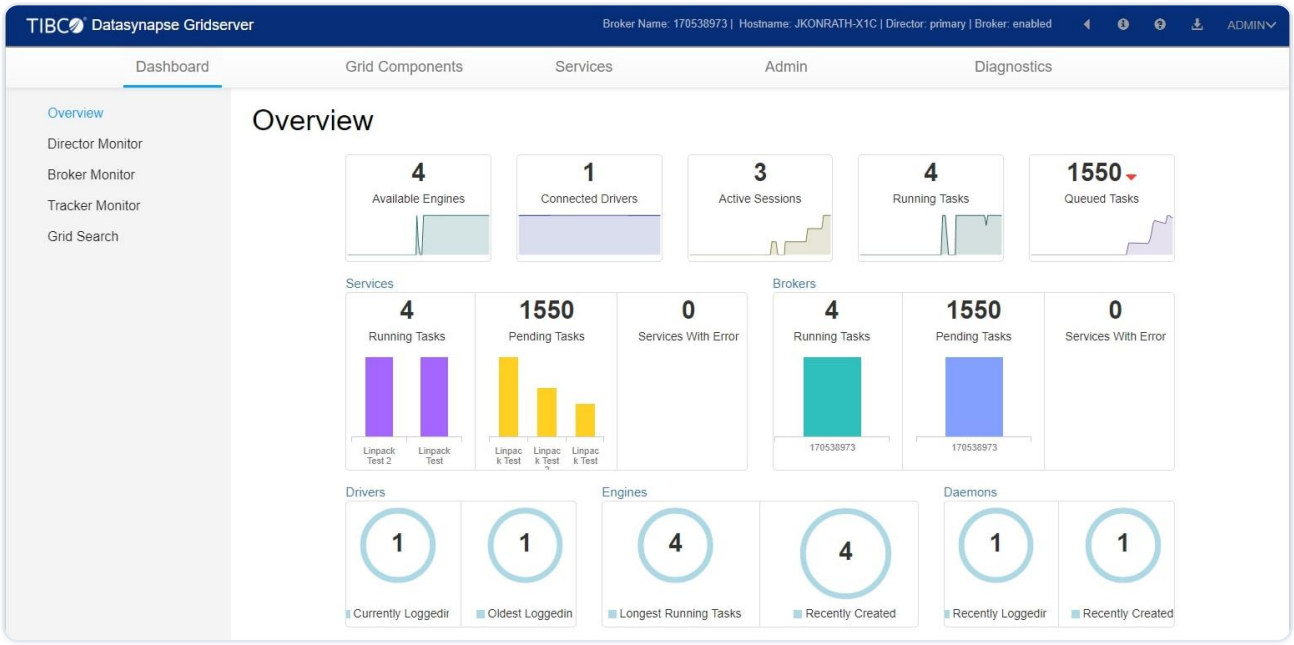
The GridServer Administration Tool

This section provides information about the GridServer Administration Tool, the web-based tool that enables the GridServer administrator to monitor and manage the Manager, its Engines, and Drivers.

Overview

The GridServer Administration Tool is a web-based tool that enables the GridServer administrator to monitor and manage the Manager, its Engines, and Drivers.

Authorized users can access the GridServer Administration Tool from any compatible browser, anywhere on the network. Administrative user accounts provide password-protected, role-based authorization.



The GridServer Administration Tool

In the GridServer Administration Tool, you can:

- Monitor Service and task execution and cancel Services
- Monitor Engine activity and restart Engines
- View and modify Manager and Engine configuration
- Download Engine installation archives
- Manage user accounts
- Subscribe to email notification of events
- Edit Engine properties and change values
- Configure routing of Drivers and Engines to Brokers
- View the GridServer API and download documentation
- Download the files necessary to develop and run GridServer applications
- Diagnose issues and monitor the grid

Getting Started

To use the GridServer Administration Tool, you must have access to the GridServe Manager from any supported browser that has JavaScript enabled.

In the browser, open `http://` *hostname* `:` *port* (where *hostname* is the address of the Manager, and *port* is the port on which it is listening, which is 8080 by default.); the Manager prompts you for a username and password.

Navigating the Administration Tool

The Administration Tool consists of a number of pages, organized in the following ways:

The Global Navigation Bar

The global navigation bar at the top of the Administration Tool contains links and information about your current login session.



The Global Navigation Bar

The following information is shown in the global navigation bar:

- The Broker name, which is a random number generated during installation.
- The Hostname of the Manager.
- If the Manager has a Primary Director, Secondary Director, or no Director.
- If the Manager has an enabled Broker, failover Broker, or no Broker.

The banner includes icons that open the following links:

- The **Information**panel, which displays component versions, built date, and applied updates.
- The **Help**panel, which displays context-sensitive help for this page, and other documentation links.
- The **Downloads**panel, which contains download links for SDKs and Engine installations.
- A **Profile**panel, which shows your current username, security roles, and has links to change your password or log out of the Administration Tool.

The Navigation Bar

The navigation bar contains categories, with each category having a submenu in the left sidebar. Each submenu has links to several pages, which are displayed in the main content area. Some submenus have subcategories at the top, which you can click to show different collections of page links.

The following categories are available:

Administration Tool Categories

MENU	CONTENTS
Dashboard	All of the sub-pages of the Dashboard.
Grid Components	Pages used to manage, view, and configure Drivers, Brokers, and Engines.
Services	Pages used to manage, view, and submit Services and Batches.
Admin	Pages used for configuration, system and user administration, and reinstallation.
Diagnostics	Pages providing logs and information useful when diagnosing issues.

When describing navigation to a page, the following format is used: **Category > Submenu > Page**. For example, to view the **Engine Admin**page, click **Grid Components > Engines > Engine Admin**.

Tables and Table Controls

Most pages have items or information shown in tables. For example, the **Grid Components > Engines > Engine Admin**page displays a table with a row for each Engine in the grid. The following controls enable you to perform actions on the listed items, or change the items that are displayed.

Action Controls

Each table item has an action control (

), which opens a list of actions for the selected items in the table. Some of these perform actions on table items, while others open a new page. If no items are selected, this acts as a Global Action control, which opens a list of actions to perform on all items.

Search Control

The **Search**control is displayed on any page containing a table. There are two types of searches: a simple search, and a query search.



The Search Control

For the simple search, enter a string in the **Search**box, select a column to search from the list, and click the search icon. A new table appears, containing the matching rows.

Click the icon next to **Search**and select **Query**for a query search. This enables you to search properties using regular expressions consisting of, at minimum, a property, operator, and value. Terms can also contain wildcards and regular expressions, as defined by `java.util.regex.Pattern`. Click **Search**to run the query.

You can enter your own complex query by selecting **Query Builder**. If you first define a query in the **Query Builder**and then select **Query**, the query is displayed in the **Find**box.

Columns Control

The **Columns**control



enables you to change the columns that are displayed in a table. When you click the Columns control, an overlay appears, listing all the columns currently in the table and columns that can be added.

To display a hidden column, click the **+**symbol next to the column in the **Add Columns**list. To hide a column, click the **X**symbol next to the column in the **Columns**list.



*The Columns overlay for the **Grid Components > Engines > Engine Admin**page*

When you add a column, it is added to the bottom of the list. The table displays the columns from left to right in the order of this list. To change the order of columns, click and drag the column name in the **Columns**list.

After you have made changes, click **Save**to apply them. To return to the default visible columns, click **Revert**.

The displayed table rows are always sorted by a column that has an arrow in it, either facing up or down. You can click this arrow to reverse the sort order of entries in the table, or click another column to change the sort column. Table order is only kept for that page view and is not persisted.

Refresh

Pages in the GridServer Administration Tool are automatically refreshed every ten seconds by default to display the most current information available. Click the **Refresh**control



to view the last time a page was refreshed, or to disable automatic refreshes. You can also customize the refresh rate by setting the `AJAX Refresh Interval` property on the **Admin > System Admin > Manager Configuration**page, in the Security section.

Pager Controls

The **Pager**controls, shown above and below tables, enable you to step through multiple pages of information, or specify how many rows appear on a page. Select a page number from the **Page**list, or select a range from the items list to display those items. You can select a greater number of

items listed per page in a table or display all of the items; type a number in the **Results Per Page**box and click **Go**.

Exporting Table Data

Most information shown in Administration Tool tables can also be accessed programmatically using the GridServer Admin API. The Admin API can also be accessed with SOAP Web Services; the WSDLs are available from the **Grid Components > Drivers > Web Services**page.

For example, to generate a list of information about all logged in Engines as normally presented on the **Engine Admin**page, you can write a simple client application that connects to the Manager and uses the `getAllEngineInfo` method of the `EngineAdmin` Web Service.

For more information about using the Admin API, see the TIBCO GridServer® Developer's Guide.

The Dashboard

The dashboard provides current information about the status of your grid, summarizing statistics into a single at-a-glance overview. The **Dashboard > Overview**page is updated automatically every ten seconds.

Status tiles on the dashboard display an indicator of that components' current state. It also displays an arrow showing the trend of that statistic, and a small graph showing the last ten minutes of the changes to that statistic. Click any status tile to open a dialog that shows more detail.

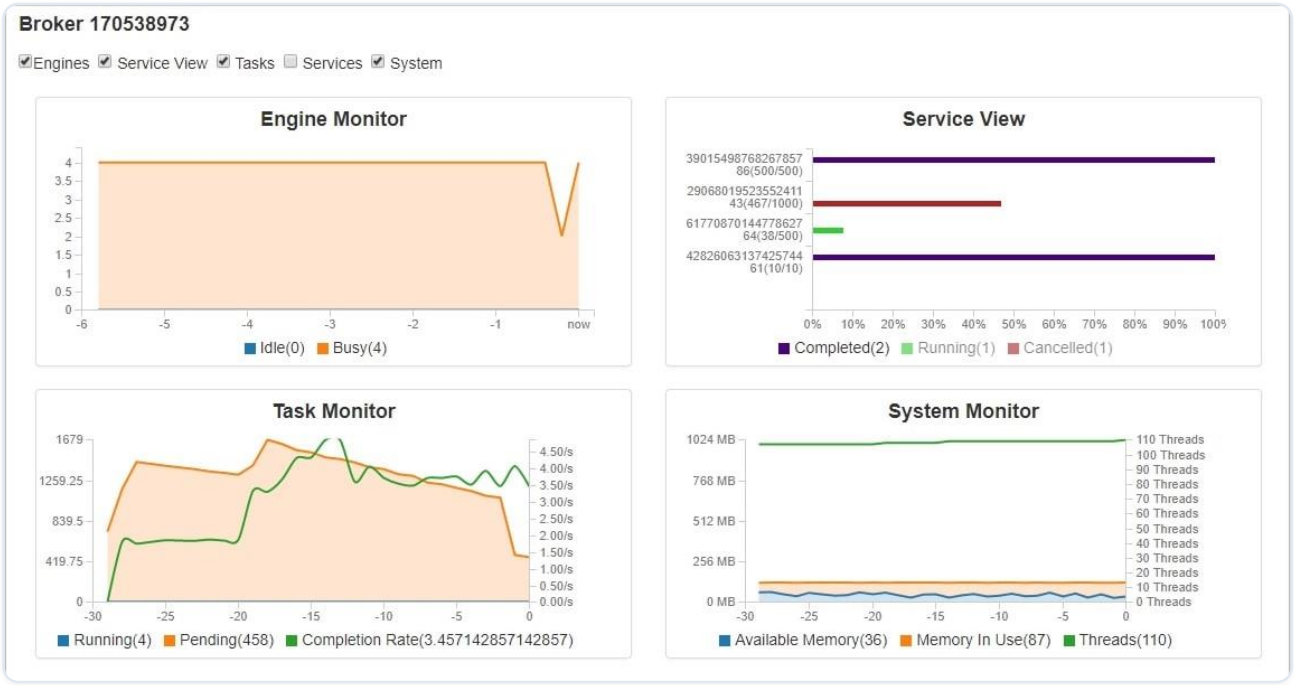
When you click the status tiles in the bottom row for grid components, the dialog also shows a small table with links to recently active components, and a link to that component's admin page. For example, if you click the Recently Created Engines status tile, it shows a table of the recently created Engines, and a link to the Engine Admin page.

The Broker and Director Monitors

The pages such as the Service Session Admin page and Engine Admin page can be used to oversee the running of Services on your grid. However, there are two graphical tools that can be used to provide a simpler overview of status information about your system. Both Directors and Brokers have a graphical monitor available, which can be displayed in its own window.

Display the Broker or Director Monitor by going to **Dashboard > Broker Monitor**or **Dashboard > Director Monitor**. Note that the Director Monitor is not available on standalone Brokers, and the Broker monitor is not available on standalone Directors.

Both monitor pages display up-to-date information about your grid. The Director Monitor contains graphs with statistics on Engines, tasks, Services and machine status, including thread and memory information. The Broker Monitor contains similar information about one specific Broker.



The Broker Monitor

User Accounts and Role-Based Access Control

To log in to the GridServer Administration Tool, you must have an account and password. GridServer supports a customizable system of role-based access control to provide account security and enable different users to access different areas of the interface.

User accounts are assigned one or more Security Roles. Each Security Role defines a set of permissions. A permission is the approval to use, see, or access a GridServer resource. There are four default Security Roles: **Configure**, **Manage**, **Service**, and **View**. The **Configure**role is for administrators and allows access to any part of the Administration Tool. By default, the root account you created at installation is set to the **Administer**role; you can also create accounts with full access for other administrative users. Users accounts assigned other Security Roles have more limited access; some pages and features either function differently, or are not available or visible.

You can create custom roles by going to the **Admin > User Admin > Role Admin**page and selecting **Create New Role**from the global actions menu. This enables you to create a new role, then add or remove what permissions are available to user accounts assigned that role. You can also use the editor to add or remove permissions from the predefined roles.

For more information about configuring user accounts and security roles, see About Authentication and Authorization.

Managing Users

GridServer user accounts enable you to specify which users are allowed to access different features within the Administration Tool, and control what resources can be used by different client software.

About Authentication and Authorization

Authentication is the process of determining if an entity is what it claims to be. GridServer provides a built-in authentication service and user repository, plus the ability to use an external LDAP server, Windows native authentication, or Kerberos.

Authorization is the process of determining what features a user has access to on a system. GridServer provides Role-Based Access Control (RBAC) to provide this, in which Users can be assigned roles manually, by using LDAP groups and by using Windows Domain groups.

Types of Users

GridServer users requiring authentication might be utilizing GridServer with client applications (Drivers), interactively using the web-based Administration Tool, or programmatically accessing the Admin interface. Each type is authenticated through the same mechanism.

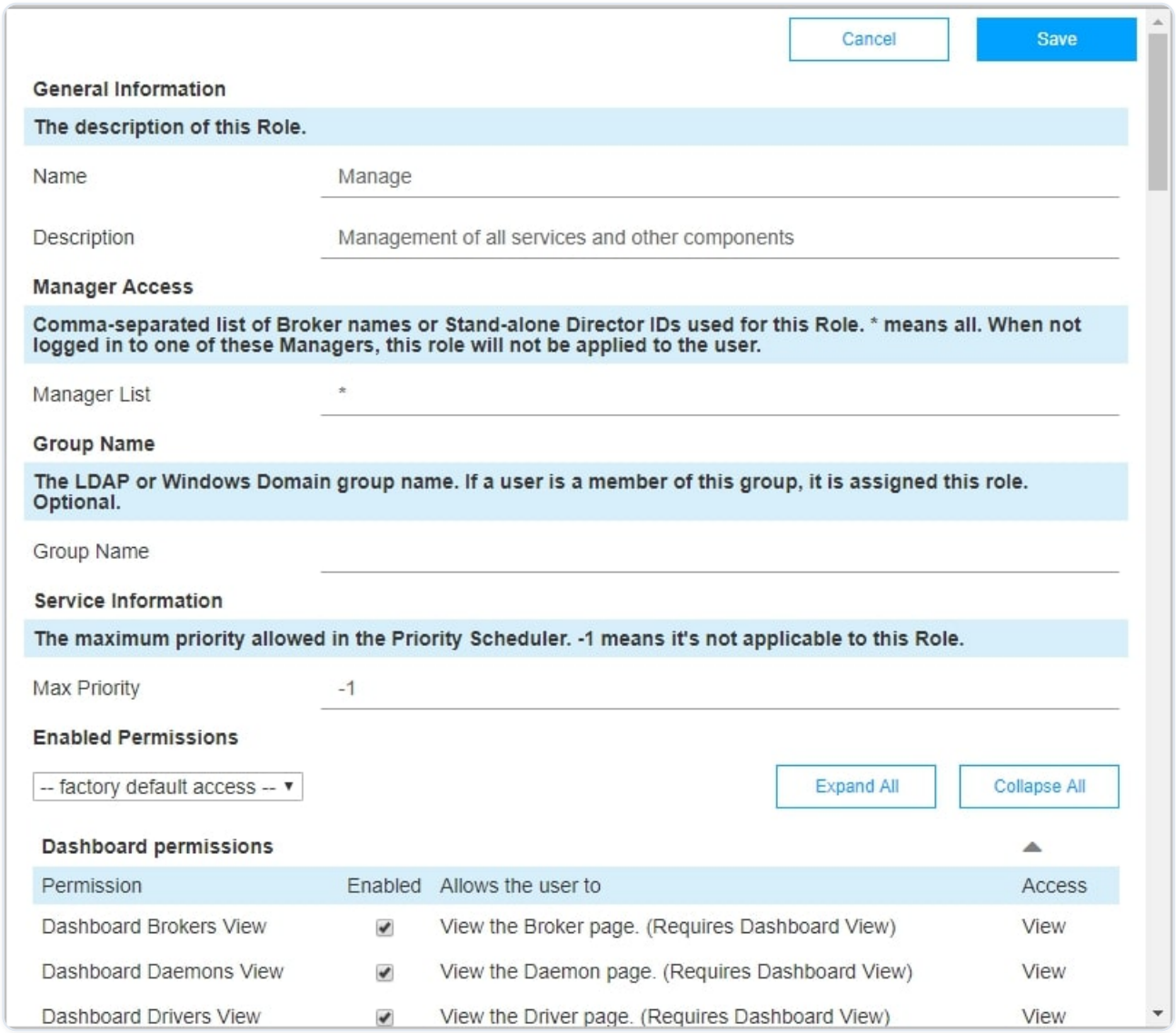
GridServer authenticates users according to the method defined by the administrator. After a grid user is authenticated, the user receives an authentication token to use in further correspondence. In the case of Administration Tool or Web Services users, the authentication token is a standard HTTP session cookie. In the case where computer users connect with the DataSynapse APIs, the authentication token is a DataSynapse object.

When a Driver attempts to log in, but has an authentication failure (incorrect user name or password), it stops trying to log in. If the failure is due to some other issue (such as an LDAP Server being temporarily down), it is not treated as an authentication failure. Any current or future service/cache methods throw an appropriate exception. The Driver process must be restarted at this point to alleviate the situation.

Operating system accounts are used to start GridServer software components, like the Manager, Engine, and Driver. It is generally not required to use a superuser operating system account to start any GridServer component.

Role-Based Access Control

Security Roles can be added and edited in the GridServer Administration Tool on the **Admin> User Admin > Role Admin**page. Each role contains a set of permissions that you can enable or disable. Each permission corresponds to a GridServer page, action, or feature. For example, you might want to disable permissions in a role to prevent a subset of users from editing Engine Daemons or managing Brokers. You can also enable permissions in a role, like enabling Service-role users to view the current Manager log. You can also use the **Role Admin**page to view what features are accessible in a given role.



Editing a Security Role

Security roles are assigned to a User via the **Admin > User Admin > User Admin**page. Also, when using LDAP or Kerberos authentication, they can be auto-assigned via LDAP groups, and when using Windows authentication, via the Windows Domain groups.

Editing Security Roles

To edit a Security Role:

1. Go to **Admin > User Admin > Role Admin**.
2. Select a role, click the **Actions**list, and select **View/Edit**.
3. Optionally, to create a new role, with no role selected, click the **Actions**list and select **Create a new Security Role**.

   The View/Edit Role page appears.
4. Select or clear the checkboxes next to the permissions you want enabled in this role. You can select another role from the **Copy**list, which selects that role's enabled features into the current role.

   You can also edit the following role attributes:

   - The name and description of the role.
   - A list of Managers on which this user can log in, or * for all Managers. If a user with this role attempts to log in to a Manager not listed, the login fails.
   - A corresponding LDAP group for the role. When an LDAP user from that group logs into GridServer, they receive this role.
   - The maximum priority a user with this role can assign to a Service.
5. When you are done editing the role, click **Save**to save changes, or **Cancel**to discard them.

The following actions are also available for each role:

- You can make a copy of a role with **Copy**.
- The **Delete**action completely removes a role. Note that you cannot delete a role if it is currently assigned to any user.

Security Role Notes

Service Session Admin methods or actions require the user to have Service Username Access to the Service in question. For example, the **Service Session Admin**page only shows a user's Services, and that user can only cancel their own Services.

Security Roles also affect the ability to use GridServer Web Services to interact with GridServer. For a list of GridServer Web Service objects and methods enabled by role, see the *GridServer Developer's Guide*.

Security Roles do not filter Services that are submitted before changing the associated Security Roles in an account. For example, a long-running Service is active and you change a user's account's Security Role association from Configure to View. In such a scenario, the user still has Configure-role access to that Service.

The Root Account Role

When a Manager is first installed, an initial account is created. This account, the *root account*, has the Root role assigned to it. The root account contains all permissions, similar to the Configure role. It is internal to the Manager, regardless of authentication mode. For example, when LDAP authentication is enabled and the LDAP server is unavailable, the root account is still available.

There are a number of restrictions related to the Root user role:

- The root account cannot be deleted.
- The root account's role cannot be changed. All other information in the root account (name, email, password, and so on) can be changed.
- You cannot add the Root role to any other account.

Using GridServer Built-In Authentication

GridServer's built-in authentication mechanism uses the embedded Director database (the internal database) to authenticate users. Passwords are stored in the database as secure hashes; when a password is entered, it is hashed and the two are compared.

GridServer built-in authentication includes options for minimum user name length, minimum password length, password complexity, password aging, and application behavior on password failure. To configure password authentication, go to **Admin > System Admin > Manager Configuration > Security**and change the settings under the **Admin User Management**heading.

Each user account is mapped to one or more Security Roles, which dictate what features of the Administration Tool they can use.

Using GridServer Built-In Authorization

GridServer's built-in role-based access control can be used to define what features a user can access on the system. Depending on the authentication type used, there are different methods for assigning roles to users.

If you are using GridServer's built-in authentication, user authorization is mapped in the user account stored within GridServer. When you create or edit an account on the **Admin > User**

Admin > User Adminpage, the **Security Roles**section of the Edit User dialog enables you to select one or more roles that are given to that user's account.

You can also use LDAP or Windows authorization, and edit roles so they map to users based on their group name. When you edit a role on the **Admin > User Admin > Role Admin**page, set the LDAP Group Name to the name of the group that receives that role.

If you are using Pure Kerberos or LDAP authentication, you can either use built-in authorization or LDAP authorization. If you use built-in authorization, this means you must create accounts for each of your users using GridServer's built-in authentication, and specify one or more roles for each user. Although you need to set passwords when you create the user, they are ignored and Kerberos is used for authentication.

Creating User Accounts

To create a User Account:

1. Log in to the Administration Tool using the Configure-role account created when you first installed GridServer, or any other account with access to the User Manage feature.
2. Go to **Admin > User Admin > User Admin**.
3. Select **Create New User**from the **Actions**list. The New User Information page appears.
4. Enter the username, a password, and confirm the password. Credentials entered here are case sensitive. If you also use Active Directory which is case-insensitive, matching credential syntax is recommended.
5. If using built-in authorization, in the **Security Roles**list, assign one or more roles by selecting the role name in the left list, then clicking the **>>**button. If multiple roles are selected, the account can access features specified in all roles.
6. You can also optionally enter a first and last name, and an E-mail address for notifications.
7. When finished entering the user information, click **Save**.

GridServer sends a notification E-mail to the address provided in the new account, provided that an SMTP host is set in the Manager's configuration. You can customize the templates used for the subject and body of these messages by selecting **Edit Email Notification Template**from the **Actions**list on the **User Admin**page.

Resetting User Accounts From the Command Line

In addition to editing a user account's password in the Administration Tool, you can also change the password of a user account in the local database from the command line on the Manager machine. This is useful if you are locked out of all accounts in the internal database.

To unlock a user account:

1. Shut down the Manager.
2. From the *DS_MANAGER* directory of the Manager install, run the `unlock.bat` or `unlock.sh` script for Windows or UNIX. The script takes two arguments: the user name and a new password. For example:

   ```
   unlock.sh jsmith NewPass123
   ```
3. Restart the Manager.

Using LDAP Authentication and Authorization

LDAP can be used for authentication, where a user is authenticated against a directory entry; authorization, where groups assigned to that user map to GridServer roles; or both authentication and authorization. For example, you can use Kerberos for authentication, but assign roles with LDAP. Or you might wish to use LDAP for authentication, but assign roles manually.

There are two authentication modes in LDAP. The *Bind Mode* authenticates a user using an LDAP bind operation (login) to the LDAP server. If the operation succeeds, the user is authenticated. In *Comparison Mode*, when a user logs in, the credentials of the user are retrieved from the LDAP server and compared to the credentials submitted in the login request. If the credentials match, the authentication is successful. Otherwise, authentication fails. In general, a hashed password is used for comparison.

Note that you do not set Bind Mode or Comparison Mode explicitly. The mode is set implicitly according to User Password Attribute: if User Password Attribute is set to any value, the authenticator uses Comparison Mode, if User Password Attribute is blank, Bind Mode is used.

If you are using LDAP for authentication, you must configure a user lookup. The User DN Format allows you to specify the user with a single parameter substitution in the DN. This is the preferred method, since it requires no LDAP search. If your directory is not configured such that you can specify this, then use the User Search Base and parameterized User Search String Format. In this case, the User Search Base specifies where to start the search (to optimize it) and the User Search String Format specifies how to match the entry by attribute.

If you are using LDAP for authorization, you must specify how the user's groups are located. If the groups are set as attributes of the user entry, use the User Group Attribute setting. Otherwise, you must use the Group Search Configuration settings.

When the group is retrieved, any role that has the Group setting set to one of these groups is assigned to the user. You can also assign groups manually, with the **Admin > User Admin> User Admin**page, instead of or in addition to assigning with LDAP.

Parameters are specified to the LDAP search using the standard format `{ n }`, where *n* is the *n*th parameter. In this case, the user name is the only parameter ever used, so use `{0}` to indicate user name.

Configuring GridServer for LDAP Authentication or Authorization

After Manager installation, when the GridServer Administration Tool is accessed for the first time, you are prompted to create a root user account. This root user account is usable regardless of whether LDAP or the internal database is used for authentication.

To configure LDAP for authentication or authorization, complete the following task to configure the connection, and then complete the tasks for configuring authentication, authorization, or both.

Configure the Connection

1. Log in to the GridServer Administration Tool using the root user account on a Manager containing the primary Director.
2. Go to **Admin > User Admin > Authentication**.
3. Select LDAP from the **Authentication Mode**list.
4. In the **Provider URL(s)**box, enter a pipe-delimited list of URLs of your LDAP servers. The first URL is the primary server and the rest are failover servers. For example, `ldap://host1:389|ldaps://host2:636`.
5. If your server does not allow anonymous search, enter values in the **Connection DN**and **Connection Password**boxes. Example values are `cn=admin,dc=company,dc=com` and `mysecret (encrypted form)`.
6. You can optionally use the **JNDI Environment Variables**box to enter a comma-delimited list of `name=value` environment variables to use when connecting to your LDAP server. For example, `com.sun.jndi.ldap.connect.timeout=500`. A list of environment variables can be found in Oracle's JNDI LDAP documentation.
7. You can also optionally configure if the host's IP address is resolved by using **Resolve IP Address**. This is false by default; if changed to true, if there is a failure connecting to the host, the IP address is logged on failures instead of the hostname. This is useful for diagnosing problem servers when using DNS load balancing.

8. To test that the values you have provided results in a working connection, click the **Test Connection**button. A Test Results popup window displays if the test script was able to connect. If not, go back to step 4 and confirm your values.

Configure Authentication

1. Configure a user lookup method:
    - For the User DN Format, specify the parameterized DN to be used to locate the user, for instance, `user={0},ou=usernames,dc=company,dc=com.`
    - For the full user search, specify the User Search Base, such as `ou=usernames,dc=company,dc=com`, and the parameterized User Search String Format, such as `(&(objectclass=user)(sAMAccountName={0}))`. This starts searching from the base for any entries that are of class user, and have an attribute called `sAMAccountName` that matches the user name. **User Search Subtree**optionally enables you to widen the scope of the user search to include subtrees. Set this to false when possible to improve search performance and reduce latency.
2. If you are using comparison mode, enter the name of the password attribute on the user entry in the **User Password Attribute**field, and select the appropriate digest method in **Password Digest**. In most cases Auto Detect is appropriate.
3. You can optionally enter attributes to retrieve other fields from your user search. These include the **User First Name Attribute**, **User Last Name Attribute**, and **User Email Attribute**boxes.

Configure Authorization

1. Configure a group lookup method:
    - If the groups are assigned as attributes of the user entry, enter that attribute name in the **User Group Attribute**box, for example, `memberOf`.
    - If a separate search is required to get the group information, you must configure the following items: In the **Group Search Format**box, enter the pattern used to match user names to group entries. For example, `(memberUid={0})` returns all groups that have a `memberUid` attribute that matches the user name. In the **Group Search Attribute**box, enter the attribute that provides the name of the group in the group entry. For example, `cn`. In the **Group Search Base**box, enter the base of a group search. For example, `ou=groups,dc=company,dc=com`.
2. Click **Save**.

To map an LDAP group to a Role, go to the **Admin > User Admin > Role Admin**page, edit the Role, and set the Group entry to the name of the group. Note that oftentimes the group name might be in CN format, especially when using group search with a cn attribute. For example, if the cn attribute for the group you want to assign to the Configure role is `CN=Administrators,CN=Builtin,DC=na,DC=tibco,DC=com,` you can edit the Configure role and specify the group as that entire name, not just `Administrators`.

Authentication Schemes Supported in Comparison Mode

When comparison mode is in use, the following LDAP server password hash/encryption schemes are supported:

Supported LDAP Schemes

SCHEME	FORMAT	DESCRIPTION	ALGORITHM	NOTES
{SCHEME} Hash	`{crypt} Q8k7rHl9JtTOI`	UNIX crypt	Calculate hash from the clear text password based on algorithm and compare with password digest	Supported by OpenLDAP
	`{SHA}! J78ElrfcxQlheAG/ XBSz76Upy5+t65mE`	SHA hashing algorithm followed by the hash		
	`{CLEARTEXT} mypassword`	Clear text password		
{SCHEME} Hash(passwd+seed)	`{SSHA}! J78ElrfcxQlheAG/ XBSz76Upy5+t65mE`	Seeded SHA hashing algorithm with the first 6 chars as the seed	Calculate the seed from the hash and calculate hash based on the clear text + salt	
PAM MD5	`$1$qPU.kEzE$SydnPwBAT0hmoK1ITsPk0`	in `$1$ [salt]>$`		

SCHEME	FORMAT	DESCRIPTION	ALGORITHM	NOTES
		`[hash]` format. `$1$` is the magic string for MD5 hashing		
Apache MD5	`$apr1$A7lJPWbr$4 1U2bOX/fj6a/`	`$apr1$[salt] $[hash]` format. `$apr1$` is the magic string		
Clear Text	`Mypassword`	clear text password in octet string, specified in rfc2256	String comparison	Must not be used

Security Notes

It's not possible for deleted users to access the Administration Tool because the GridServer Director controls all user add/update/delete operations. When a user is deleted or demoted to a group with lower privileges, all other GridServer Managers get the user table update and refresh their local user cache.

When LDAP is used, this is no longer possible, as GridServer won't get notifications for LDAP user updates. Therefore, a timeout strategy is used to revalidate the user authorization. User authorization has a 15 minute TTL that is independent of the Application Server session. A deleted/demoted user does not have indefinite access to the features that are no longer permitted.

LDAP Configuration Example

For a typical example configuration, consider an OpenLDAP server that can authenticate in both Windows and Linux/UNIX domains. In the test LDAP schema, group information is specified in individual group searches.

First, go to Role Administration and set up group names for the roles. Map the Manage role to the support LDAP group.

Then configure the connection to the LDAP server as follows:

- Authentication Mode = `LDAP`
- Provider URL(s) = `ldap://integrated.datasynapse.com:389`
- Authentication Scheme = `simple`

Leave the **User name**and **Password**blank since this directory allows anonymous search.

Next, after successfully testing the connection, configure the user search by using the user DN, and leaving Search String and Search Base blank:

- User DN Format = `uid={0},ou=users,dc=datasynapse,dc=com`
- User Search String Format = not set
- User Search Base = not set
- User Search Subtree = `False`
- User Search Timeout = `5000`
- User Password Attribute = `userPassword`
- User Password Digest = `Auto Detect`

Finally, set up the group search. Since this LDAP schema keeps the groups as separate entities, leave User Group Attribute blank, and specify a separate group search:

- Group Search Attribute = `cn`
- Group Search Base = `ou=groups,dc=datasynapse,dc=com`
- Group Search Format = `memberUid={0}`
- Group Search Limit = `0`
- Group Search Timeout = `5000`
- Group Search Subtree = `True`

Using Windows Authentication and Authorization

Another method of user authentication is *Windows Authentication*. This takes advantage of the native Windows authentication layer when the Manager is running on a Windows machine on a Windows Domain. It uses the Negotiate protocol, like *Pure Kerberos Authentication*, but requires much less configuration. It uses NTLM or Kerberos as the authentication provider, depending on the client; Windows C++ and .NET use NTLM, while other clients use Kerberos. This enables web browsers and Drivers to connect to Managers by authenticating as the current Domain user, with no passwords needed.

Also, all Domain groups that a user belongs to are available as a group that can be mapped to a GridServer Role.

Configuring Windows Authentication

To use Windows authentication, you must configure your Manager, Drivers, and web browsers. Note that you can only use this method if all Managers run on Windows.

Manager Configuration

To configure your Manager to use Windows authentication:

1. Ensure that the Manager is part of the domain with which you want to authenticate.
2. Make sure that the additional third-party LGPL download has been applied. For more information, see the *Installation Guide*.
3. On the Managers:

   In the Administration Tool, go to **Admin > User Admin> Authentication**, and change **Authentication Mode**to **Windows**.

   On the same page, enter the value for the Windows Domain used to authenticate users in the Windows Domain, and click **Save**.

Driver Configuration

For all Drivers, the `DSNegotiateEnabled` property specifies if Negotiate authentication is used. Set this to true in the `driver.properties` file or by using the `DriverManager` API to enable Negotiate authentication.

Also, the following drivers and browsers must be configured, depending on the platform and Driver:

Windows C++ and .NET Drivers

Windows versions of C++ and .NET Drivers use NTLM and do not require any additional configuration.

UNIX C++ and All Java Drivers

UNIX versions of C++ and all Java Drivers (including Windows) use Kerberos rather than NTLM. Configure them using the instructions in All Java Drivers and UNIX C++ and Java Drivers sections.

Browser Configuration

Users' browsers must be configured to use Negotiate authentication. For example, in Microsoft Edge, you add the URL to the Enterprise Mode site list. In Firefox, you use the `network.negotiate-auth.trusted-uris` config parameter. See your browser's documentation for details.

If a user's browser is not configured and they attempt to log in to the Administration Tool, the browser presents them with a challenge popup screen, and they can log in manually.

Configuring Windows Authorization

When a user logs in, the list of Domain groups that user belongs to is available. All roles that are assigned to those groups are then assigned to the user.

Note that all the users must be assigned to a group that is mapped to at least one role to be able to log in. Additional roles might be mapped with Windows groups or built-in authorization.

Using Pure Kerberos Authentication

If you are not running all Managers on Windows, and you require authentication that does not require a password, you use *Pure Kerberos* authentication. This also enables web browsers and Drivers to connect to Managers by authenticating with a Windows domain, and Drivers no longer have to store passwords.

The section assumes that you already have a Kerberos Realm set up, with a Kerberos Key Distribution Center (KDC). Since the most common Kerberos installation is a Windows Domain on which the KDC is on a Domain Controller (DC), this section proceeds with that as an example.

Configuring Pure Kerberos Authentication

The following is an overview of the process:

On each Manager, there is a user that is a client of the Key Distribution Center (KDC) on the DC. This must be a user on the Windows domain. This can be an existing user, or a new user can be created. Also, this user does not need to be the same user who runs the Manager process, although it can be.

A Service Principal Name (SPN) is created on the KDC. This is the name by which a client uniquely identifies an instance of a service. It consists of the user name and Manager hostname. There might be more than one per Manager; for example, one for the fully-qualified hostname, one for the short name, and additional ones for aliases if any.

A keytab file is created for the user and placed on the Manager's file system. It contains the user credentials, and allows that user to use the KDC.

The Manager is then configured to use the realm, given the keytab location and DC hostname.

Note that it is not necessary for the machine to be on the domain. While it is true for a Windows Manager, there is no requirement for a UNIX Manager to be added to the domain.

To configure your Manager, you need the following information from your IT department:

- The name of your realm
- The fully-qualified hostname of the DC

The following table lists the various values that are used in the following procedure. Substitute your own values for the following:

SETTING	VALUE
Realm Name	*REALM.NAME*
DC Fully Qualified Hostname	*dc.domain.com*
Manager Fully Qualified Hostname	*manager.domain.com*
GridServer Username	*gs_user*

Manager Configuration

To configure your Manager for pure Kerberos authentication:

1. Map one or more Server Principal Names. Log in to *dc.domain.com*. Open a console, and run the following to create SPNs for the short and fully qualified names:

   ```
   setspn -A HTTP/manager gs_user

   setspn -A HTTP/manager.domain.com gs_user
   ```

2. Create the keytab file:

   From that same DC, generate a keytab:

```
ktpass \

/princ HTTP/manager.domain.com@REALM.NAME \

/ptype KRB5_NT_PRINCIPAL \

/crypto all \

/mapuser gs_user@REALM.NAME

/pass {gs_user password}\

/kvno 0\

/out GridServerUser.keytab \
```

Move this file to your Manager, and place it in a location and set permissions such that only the user that runs the Manager can read it. Because it contains credential information, it must be kept secure.

3. The KDC client requires a random sequence to protect the session. You can generate one using the openssl command line tool by running the following command:

   `openssl rand -hex 12`

4. Edit the DS_MANAGER `/webapps/livecluster/WEB-INF/web.xml` file:

   Uncomment the `kerberosFilter` section.

   Set the param-value of `kerberos.principal` to `HTTP/`` manager.domain.com@REALM.NAME`

   Set the param-value of `kerberos.keytab` to the location of the keytab file on your Manager.

   Set the param-value of `signature.secret` to the random sequence that was just generated.

Driver Configuration

For all Drivers, the `DSNegotiateEnabled` property specifies if Negotiate authentication is used. Set this to true in the `driver.properties` file or by using the `DriverManager` API to enable Negotiate authentication. Otherwise username and password is used.

Also, the following platform or drivers must be configured, depending on the platform and Driver:

All Windows

- You must enable TGT on the system running the Driver. For more information, see https://support.microsoft.com/en-us/topic/updates-to-tgt-delegation-across-incoming-trusts-in-windows-server-1a6632ac-1599-0a7c-550a-a754796c291e.
- The `allowtgtsessionkey` registry key must be set. Add or change this registry value:

  `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters`

  Value Name: `allowtgtsessionkey`

  Value Type: `REG_SZ`

  Value Data: `1`
- The Driver user cannot be a member of the Local Admin group.

All Java Drivers

Java includes support for Kerberos authentication natively. The standard method of doing this is via the `login.conf` and `krb5.conf` files.

Convenience properties have also been added to the `driver.properties` file so that you do not need to set these files up:

- `DSKerberosRealm`: the name of the kerberos realm. For example, `REALM.NAME`
- `DSKerberosKDC`: The hostname of the KDC. For example, `dc.domain.com`

(Note that if your application is already set up to use Kerberos via standard methods, you do not need to set these two values.)

Windows C++ and .NET Drivers

Create a file called `krb5.ini` and put it in the `c:\windows` directory:

This file must contain the following:

```
[libdefaults]

default_tgs_enctypes = AES256-CTS RC4-HMAC DES-CBC-MD5 DES-CBC-CRC

default_tkt_enctypes = AES256-CTS RC4-HMAC DES-CBC-MD5 DES-CBC-CRC

preferred_enctypes = AES256-CTS RC4-HMAC DES-CBC-MD5 DES-CBC-CRC
```

```
[domain_realm].domain.com = REALM.NAME

dc = REALM.NAME

dc.domain.com = REALM.NAME
```

UNIX C++ and Java Drivers

UNIX Kerberos clients use a ticket cache. This ticket cache must be populated prior to running the Driver.

This is done by running the `kinit` command, which prompts the user for their password, and populates the cache with a ticket.

Browser Configuration

Users' browsers must be configured to use Negotiate authentication. For example, in Microsoft Edge, you add the URL to the Enterprise Mode site list. In Firefox, you use the `network.negotiate-auth.trusted-uris` config parameter. See your browser's documentation for details.

If a user's browser is not configured and they attempt to log in to the Administration Tool, the browser presents them with a challenge popup screen, and they can log in manually.

Configuring Pure Kerberos Authorization

Because Kerberos only provides authentication, authorization is provided with LDAP or built-in authorization.

To be able to log in, all users must belong to at least one role that is assigned to a Windows Domain group, which corresponds to an Active Directory LDAP group. See, Using LDAP Authentication and Authorization. You only need to configure Authorization. Additional Roles might be mapped with LDAP groups or built-in authorization.

Managing Multiple Brokers with Grid Single Sign-On (SSO)

The Grid Single Sign-On (SSO) feature enables a user to be logged in to all Managers on a Grid after having logged in once to any Manager.

> **Note:**

> The use of Grid SSO is discouraged; using Kerberos with LDAP or Windows authentication is preferred to achieve single sign-on. Grid SSO must never be used when Pure Kerberos or Windows Authentication are enabled.

> The following prerequisites are required before you can enable Broker SSO:

- All Managers must reside in the same network sub-domain
- The Primary Director's hostname that appears in the **Admin > System Admin > Manager Reinstall > Local Configuration > Hostname**field must be a fully-qualified name in this sub-domain.

Grid SSO Configuration

To configure Grid Single Sign-On:

1. Log in to the GridServer Administration Tool on the Manager containing the Primary Director.
2. Go to **Admin > System Admin > Manager Configuration > Security**.
3. Click the **Security**link.
4. Set **Grid Single Sign-On sub-domain**to the network subdomain used by your Managers. At installation, GridServer attempts to determine the subdomain name based on the fully qualified host name of the Director, and the value is entered in the **Grid Single Sign-On sub-domain**field. If this value is incorrect, you must change it before enabling SSO.
5. Set **Grid Single Sign-On**to `true`. Note that you must configure all Directors and Brokers with their fully qualified hostnames.
6. Repeat this configuration for each Manager containing a Director in your grid. You do not need to configure Managers with only Brokers.

Constraints and Limitations

The following constraints apply when using Grid SSO:

- You must enable cookies in your browser.
- When a Primary Director fails, the Grid SSO session is lost. If you have a Secondary Director configured for failover, you can log in to it and start a new SSO session.
- Single sign-on enables you to log in to other Managers that are in the same network subdomain. For example, if your Primary Director is `director.grid.example.com`, logins to any Brokers in `grid.example.com` can be authenticated with Grid SSO. You can't use Grid SSO to connect to Brokers in other subdomains or domains. For example, if your Primary

Director is `director.grid.example.com`, you can't automatically log in to Brokers in `prod.example.com` or `grid.example.co.uk`.

- Only the following fields accept multi-byte characters:
- **Username**, **Password**, and **Confirm Password**on the initial log in page.
- All fields on the **Admin > User Admin > Run-As Credentials**page.
- **Choose Username**on the User Event page.
- **Username**, **Firstname**, **Lastname**, **Password**, and **Confirm Password**on the **Users > Users**page.
- **RunAsUser**on the Edit Service Type page.
- Grid SSO must not be used if Negotiate Authentication is used.

Client Routing

You can route Clients to Brokers by using Roles, the Broker Routing page, and the Driver API.

Routing Clients With Roles

The easiest and most common method of routing clients is to set the Manager List in a user role, and then set a user name in the `driver.properties` file. This routes clients directly to a set of listed Brokers.

To configure the Manager List:

1. In the GridServer Administration Tool, go to **Admin > User Admin > Role Admin**.
2. Select a role, click the **Actions**list, and select **View/Edit**, or to create a new role, with no role selected, click the **Actions**list and select **Create a new Security Role**.
3. Enter the list of Brokers to which you want the client to be routed.
4. Click **Save**.
5. Go to **Admin > User Admin > User Admin**.
6. Select a user, click the **Actions**list, and select **Edit User**, or to create a new user, with no role selected, click the **Actions**list and select **Create New User**.
7. Assign the role to the user.
8. Edit the `driver.properties` file for the Driver, and set the `DSUsername` and `DSPassword` properties to the user name and password of the user assigned the role you created/edited.

Routing Clients On The Broker Routing Page

You can route clients to Brokers using rules based on Driver properties. This is similar to how you can route Engines to Brokers by creating routing rules based on user-defined properties. This limits Drivers that log in to a Broker based on the value of one or more properties, using comparators you have defined.

To enable Driver Routing:

1. Define one or more properties in the `driver.properties` file used by the Driver. You can view a Driver's properties at the **Grid Components > Drivers > Driver Admin**page.
2. Go to the **Grid Components > Brokers > Broker Routing**page. Select a Broker from the list.
3. Click **Edit**.
4. In the Driver column are comparators for Drivers. Enter one or more comparators by entering a property name, selecting an operator, entering a value, then clicking **Add**. You can also select **Is Missing**, which evaluates the comparator as true if the property is not present on a Driver.

Multiple comparators are ANDed together. If the comparators evaluate to true, the Driver is allowed to log in to the Broker. Note that property names and values are case-sensitive.

You can add a single property in a comparator that holds different values by using a comma-delimited value. For example, Broker A can have a comparator matching a `username` property to be equal to `TestUser1` or `TestUser2` . If a Driver submits with either `TestUser1` or `TestUser2` values, the Tasks are then routed to Broker A. In this use case, the query must be: `username equals TestUser1,TestUser2`.

Routing Clients With The Driver API

You can use the `connect(String broker)` method of the `DriverManager` API on all Driver platforms to force a client to log in to a specified Broker.

Managing Services

This section provides information about deploying, running, and scheduling Services.

Deploying Services

GridServer uses *Grid Libraries* to distribute classes, libraries, and other resources to Engines. Grid Libraries provide a solution to the problem of distinct Services requiring different versions of the same resource. They provide the following features:

- Version control, including optional automatic selection of the most current version of a Grid Library.
- Resource upgrading without interrupting current Sessions.
- Specification of dependencies on other Grid Libraries.
- Specification of which C++ runtime to use and non-default JREs via dependencies.
- All-in-one packaging for JAR files, native libraries for multiple OSes, .NET assemblies, Command Service executables, R scripts, and Engine Hooks.
- Specification of Environment Variables and Java System properties.
- Engines that require different compiler support libraries can participate in the same Service Session.
- Parameterization of package configuration through the use of property substitution files.

The *Resource Deployment* feature replicates Grid Libraries from a Manager to Engines. In the simplest sense, this enables you to copy a set of bundled resources to each Engine to run a Service.

This section details how to use Grid Libraries for Service deployment to your GridServer installation.

About Grid Libraries

A Grid Library is a set of resources and properties necessary to run a Grid Service. It also contains configuration information that describes how these resources are to be used. For example, a Grid Library can contain JAR files, native libraries, configuration files, environment variables, hooks, and other resources.

A Grid Library is deployed as an archive file in ZIP or gzipped TAR format, with a `grid-library.xml` file in the root that describes the Grid Library. It might also contain any number of directories that contain resources.

Grid Libraries are identified by name and version. Versions are optional, but recommended. They are used to detect conflicts between a desired library and a library that has already been loaded. Versions also provide for automatic selection of the latest version of a library. A GridServer Service can specify that it is implemented by a particular Grid Library when its Service Type is registered, or by using a Service Option.

Grid Libraries can specify that they depend on other Grid Libraries. Like the Service Option, such dependencies can be specified by the name, and optionally the version. Also, nearly all aspects of a Grid Library can be specified to be valid only for a specific operating system. This means that the

same Grid Library can specify distinct paths and properties for Windows and Linux. However, only the appropriate set of package options is applied at run time.

Variable Substitution

You can use placeholder variables in a `grid-library.xml` file, which are then substituted with their value as defined in a properties file or in an OS environment variable. This enables quick changes in properties in the `grid-library.xml` file without redeploying the Grid Library.

If the `grid-library.xml` file contains a property with a value contained with the `$` character, such as `$mydir$`, it is substituted with the value in one of three places, in this order:

- A default properties file in your Grid Library named `grid-library.properties`. This can provide baseline values for your variables.
- An external properties file, named with the same name as the Grid Library archive, with the extension `.properties`, in the Grid Library deployment directory. Values in an external properties file replace those defined in the default properties file within the Grid Library.
- A defined OS environment variable. This value replaces the value defined in either properties file.

> **Note:**

> If the substitution is not found in the file, the empty string, "", is substituted.

> Substitutions are allowed anywhere in a string within the content of property value elements and path elements. Multiple substitutions per string are allowed. `$` characters can be treated as literals by escaping them with another `$` character. Windows paths that are specified in the *library* `.properties` file must escape the `\` character with another `\`.

Versioning

Versioning provides the following functionality:

- It allows for deployment of new versions of libraries and deletion of old versions without interrupting currently running Service Sessions.
- It provides for specifying conflicts, or libraries that cannot coexist with each other.
- It allows for a Service Session or dependency to specify the use of the latest version of a Grid Library.

To use versioning, you must specify the Grid Library version in the configuration file. An Engine can load only one version of the library with the same name at any time. If the version is not specified, it is implied to be 0.

The version is a String and **must**adhere to the following version format. This format is

```
[n1].[n2].[n3]...
```

where `nx` is an integer, and there might be one or more version points.

For instance,

```
4.0.1.1,  4.1,  3
```

are in the proper comparable version format.

The integer at each version point is evaluated starting at the first point, and continues until a version point is greater than the other. If a version point does not exist for one, it is implied as zero.

For instance

```
4.0.0.1 > 4.0

4.0.0.5 < 4.0.1.1
```

To specify that a dependency or Service uses a particular version of a Grid Library, the version field is set to that value. To specify that it uses the latest version, the field is left blank.

Note:

If a version is specified, it must match exactly. That is, `3.0.0` is not the same as `3`. If the library's version is `3.0.0` and the Service specifies `3`, the Service does not find that library and then it fails.

If a version is specified but not in this format, and there are multiple versions of a library, the "latest version" is undefined. Thus, automatic selection of the latest version is only possible when all Grid Libraries with the specified name provide a version in the proper format.

By default, if a Service has been set to use the latest version of a Grid Library, all Engines work on the latest version at the time the Service is started. This is regardless of whether a newer library has been deployed. This can be changed by setting the `GRID_LIBRARY_STRICT_VERSIONING` option in the `driver.properties` file to false. When false, if you deploy a newer version of the library while the Service is running, Engines that have not yet worked on the Service use the newer version. Engines that worked on it before deployment continue to use the older version.

Dependencies

Grid Libraries might specify dependencies on other Grid Libraries. A dependency specification resolves to a particular Grid Library using two values:

- **grid-library-name** - the name of the Grid Library, as specified in the dependency's XML
- **grid-library-version** - the version of the Grid Library, as specified in the dependency's XML. OS compatibility is determined by checking the `os` and `compiler` tags for the top-level element in the dependent Grid Library. If not specified, it uses the latest version supported by the OS

Note that if a dependency resolves to more than one Grid Library, the dependency used is undefined.

Two dependent libraries conflict if they have the same library name, but different versions.

It is possible to specify an OS attribute to a `<dependency>` element for ignoring Grid Libraries that do not apply to an Engine's particular operating system. For example, if a Grid Library contains native libraries for multiple platforms, you can specify OS-specific dependencies on the bridge Grid Libraries such that the Engine only loads the bridge corresponding to its operating system.

Note that if a dependency is missing, the Engine logs a warning. Rather than the current task failing, the Engine attempts to continue loading all necessary libraries to run the task.

Conflicts

A conflict between two Grid Libraries means that these libraries cannot be loaded concurrently. When there is a conflict between a loaded Grid Library and a Grid Library that is required by a Service, the Engine must restart to unload the current libraries and load the requested library.

The following circumstances result in a conflict:

- **Version Conflict** - the most common conflict arises with versioning, and typically when upgrading versions or using more than one version of the same library concurrently. This

conflict arises when a Grid Library with the same `grid-library-name` as the requested Grid Library, but a different version, is loaded.

- **Explicit Conflict** - there can be situations in which different Grid Libraries can conflict with each other due to conflicting native libraries, different versions of Java classes, and so on. Because the Engine cannot determine these implicitly, the `conflict` element can be used to specify Grid Libraries that are known to conflict with this Grid Library.

  Additionally, the value of the `grid-library-name` can be set to `"*"`. This means that this Grid Library can conflict with all other Grid Libraries (aside from its dependencies), and it is guaranteed that no other Grid Libraries load concurrently with this Grid Library. Note that this is only allowed if the Grid Library is not a dependency. If the `"*"` is used as a conflict in a Grid Library that is a dependency, a verification error occurs.

- **Dynamic Version Conflict** - a Grid Library conflict occurs if:

1. Dynamic versioning is used.
2. The latest version of a Grid Library or Grid Library dependency has changed due to an addition or removal of a dependency since the Grid Library has been loaded.

    - **Variable Substitution Conflict** - a Grid Library conflict occurs if its variable substitution file has changed since it has been loaded.

JNI Library Loading

When a Service Session is set to use a Grid Library, that library is loaded. Loading is the process of setting up all resources in the Grid Library for use by the Service. A library is loaded only once per Engine session.

First, the library loads itself, and then it loads all dependencies. The library loader uses the depth-first, or preorder traversal algorithm when loading libraries. When there are a number of dependencies in a Grid Library, the order in the XML is considered left-to-right with respect to the algorithm. The library search order for `lib-path` and `jar-path` is determined by their respective lists. Certain aspects of a load might require a restart, and possibly re-initialization of the state.

The following steps are performed by a load of the root library and all dependencies:

1. It checks for conflicts with currently loaded Grid Libraries. If so, it restarts with the requested Grid Library and clears out the current state of any loaded libraries.

2. If new `lib-path` s have been added for its OS, they append to the current list of `lib-path` s. The state of loaded libraries includes all libraries already loaded, plus the requested library. Note that specifying a JRE dependency has this effect.
3. If new `jar-path` s have been added for its OS, the JAR files and classes are added to the classloader.
4. If new `assembly-path` s have been added, it adds them to the .NET search path.
5. If new `command-path` s have been added for its OS, it is added to the search path for Command tasks.
6. If new `hooks-path` s have been added, any hooks in the path are initialized.
7. If the default is current and a Grid Library is requested, the Engine restarts.

State Preservation

Under most cases, when an Engine shuts down, it preserves the current state of which Grid Libraries it has loaded. When it starts back up, it loads all Grid Libraries that were loaded when it shut down. As Grid Libraries are loaded, the path elements they contain are added to a "master" list of paths for that type of path element. For example, if a Grid Library contains a `lib-path` specification, that `lib-path` is appended to the list of `lib-path` values obtained from already-loaded Grid Libraries.

Note that this means that it is up to the creator of the Grid Libraries, which are deployed on the grid, to ensure that the ordering of library paths does not lead to loading the wrong library. For example, consider two different Grid Libraries each provide DLLs in their `lib-path` s that share the name. Because of OS-specific library loading conventions, the one used is the first one in the aggregate `lib-path` from across all loaded Grid Libraries. Likewise, for Java classes, when more than one copy of the same class is in the classloader, it is undefined which class loads. Therefore, it is important to either subdivide Grid Libraries appropriately when there is a possibility that such conflicts can arise. You can also use the `conflict` element to explicitly state conflicts.

Grid Library and RunAs State information persists on normal Engine shutdowns. It includes task failures aside from crashes. If the Engine does not shut down normally, such as if it crashes, or if the Daemon kills the process due to it exceeding the shutdown timeout, the state is reset.

If an Engine shuts down due to a conflict, it clears the current state and sets up for only the requested Grid Library on restart. This is referred to as preloading. If an Engine shuts down due to internal library inconsistencies or a crash, the state is not saved. State is also cleared on all instances when a Daemon is disabled and re-enabled.

Task Reservation

If an Engine requires a restart to load a Grid Library, the task is reserved on the Broker for that Engine. The Engine is instructed to log back into the same Broker, and takes that task on log in. You can condfigure the timeout for this at **Admin > System Admin > Manager Configuration > Services**.

Environment Variables and System Properties

All environment variables and Java System properties for a Grid Library and all dependencies are set each time a task is taken from a particular service that specified that Grid Library. (They are not cleared after the task is finished.) Environment variables are set via JNI so that they can be used by native libraries or .NET assemblies. They are also passed into Command Services. Note that environment variables such as `PATH` and `LD_LIBRARY_PATH` must not be changed through this mechanism. Rather, `library-path` and `command-path` are reserved for manipulating these variables.

Using Grid Libraries from a Service

Services can specify a Grid Library to use by setting the `GRID_LIBRARY` and optionally the `GRID_LIBRARY_VERSION` Service Options. This is typically set by Service Type in the **Services > Services > Service Types**page, although it can be set programmatically on the Session. Services can specify a Grid Library to use by setting the corresponding Service Option values. If the version is not set, a Service uses the latest version of a Grid Library.

If a Service needs to find resources in a Grid Library, it can use the Grid Library Path. This value is a path value that includes the root directories of all Grid Libraries currently loaded. For Java, .NET, and C++, the path is `EngineProperties.GRID_LIBRARY_DIR;` for command services, it is the environment variable `ds_GridLibraryPath`.

Super Grid Libraries

A Grid Library can be declared as a Super Grid Library. This means that it is always loaded when the Engine starts up. The typical use case for this is to have an `EngineHook` that queries the system for some information, which is used to set `EngineSession` properties prior to the Engine running any tasks.

To specify that a Grid Library is a Super Grid Library, set the `super` attribute in the `grid-library` element. For example, `<grid-library .. super="true" />` . Super Grid Libraries also cannot have conflicts or dependencies. Other libraries cannot depend on or conflict with them.

Super Grid Libraries are loaded upon startup before anything else. They are ignored on conflict checks for `*` (all).

If a new Super Grid Library is deployed while an Engine is running, it is loaded. If a new version of an existing Super Grid Library is deployed while an Engine is running, the Engine restarts.

Deployment

Grid Libraries are deployed using the resource deployment page at **Services > Services > Grid Libraries**, or by using the Admin API. When a Grid Library is uploaded, it is first verified to ensure that the ZIP archive is not corrupt, and that the `grid-library.xml` file validates against the Grid Library DTD. If there is an error, it is displayed next to the file on the **Services > Services > Grid Libraries**page.

The Resource Manager then replicates uploaded, valid libraries to all Engines. Variable Substitution property files also must be placed in this directory. Engines download Grid Libraries based on the attributes in the root level `grid-library` element. Grid Libraries whose attributes match the properties of a particular Engine are downloaded by that Engine and ignored by Engines with non-matching properties. If no attributes are specified in this element for a particular Grid Library, all Engines download it.

Adding or removing Grid Libraries or property files do not trigger an Engine and Daemon restart. It is not necessary to restart until the Engine actually needs to use the Grid Library, and even then only if necessary according to the loading procedure. If a deployed Grid Library is changed, it does cause the Daemon and Engines to restart. Also, it is the responsibility of the user not to delete Grid Libraries loaded by active Services from the Libraries page, as that might lead to library load failures for tasks that are run later.

If you are not using the Resource Manager for replication, you can use an alternate shared Grid Library directory. Then set the **Grid Library Path**in all Engine Configurations to point to this directory, instead of the default replicated location. When changes are made to this library, you must then use the **Update**button on the **Services > Services > Grid Libraries**page on the Primary Director. This sends a message to all Engines to check and update their Grid Libraries using the Grid Library Manager.

The Resource Manager uses secure hashes as file signatures when determining if a file has changed on the Manager. All files are signed when the Manager starts. After that, a file is only signed again if the file’s last modified time has changed since the sign.

Disabling Resource Deployment

There are some situations in which you might not want to use the Resource Manager for replication. For example, you might want to use another shared location for Grid Libraries, deploy resources manually, or deploy different sets of resources to different Brokers.

There are four different strategies for disabling resource deployment:

- Disabling Director to Broker synchronization on the Director.
- Disabling Director to Broker synchronization on the Broker.
- Disabling Broker to Engine synchronization on the Broker.
- Disabling Broker to Engine synchronization in an Engine Configuration.

Each of the strategies is described below.

Note that for manual deployment, when you deploy Grid Libraries, you must click the **Update**button on the **Services > Services > Grid Libraries**page to notify all Engines to rescan their directories.

Disabling Director to Broker Synchronization on the Director

If you want to have different sets of resources on different Brokers, you can disable the Director to Broker resource synchronization on the Director. This causes none of the Brokers reporting to that Director to synchronize resources with that Director. The Brokers continue to synchronize resources with their Engines.

To disable Director to Broker synchronization on the Director:

1. Go to **Admin > System Admin > Manager Configuration > Resource Deployment**.
2. Under the **Director Settings**heading, change **Synchronize Resources To All Brokers**to **False**.

When you disable Director to Broker Synchronization, if for any reasons your Engines move to a different Broker, they need to synchronize their resources with the new Broker. This can result in a significant delay in the Engine being able to take tasks and can cause a severe increase in network traffic depending on how many Engines move.

> **Note:**
>
> Grids that rely on Engines being shared are strongly recommended not to use this option. If you must use this option, minimize Engine movement by disabling Engine sharing and

> balancing. In general, we do not recommend you use this option and consider Grid Library filters instead.

Disabling Director to Broker Synchronization on the Broker

In some situations, it might be desirable to disable Director to Broker resource synchronization on specific Brokers, while still permitting Broker to Engine resource synchronization. For example, allowing "last week's" tasks to run to completion on a limited subset of the grid, while permitting "this week's" tasks to be launched with new resources.

This is similar to disabling synchronization on the Director, but because the setting is on the Broker, it is used on a case-by-case basis. You can have some Brokers synchronize, and disable synchronization on others.

To disable Director to Broker synchronization on a Broker:

1. Go to **Admin > System Admin > Manager Configuration > Resource Deployment**.
2. Under the **Broker Settings**heading, change the value of **Synchronize Resources From Director**to `False`.

> **Note:**
>
> If you disable synchronization on the Director, no Brokers synchronize with the Director regardless of this setting on the Broker.

Disabling Broker to Engine Synchronization on the Broker

If you want to deploy resources manually to Engines, such as when you use a shared resource location on a shared file system, you can disable synchronization between a Broker and all of its Engines.

To manually deploy resources:

1. Go to **Admin > System Admin > Manager Configuration > Resource Deployment**, and under the **Broker Settings**heading, change the value of **Synchronize Resources to Engines**to `False`.
2. Unpack the required Grid Libraries in a shared location, such as a shared network drive that the Engines can access. Manually extract the resource files, as the Engine does not unpack them.

3. In the Engine Configuration, set **Grid Library Path**to the location of the shared resources from step 2.

> **Note:**
>
> When Broker to Engine synchronization is disabled, Director to Broker synchronization still occurs. This means that the Engines still auto-upgrade themselves when an update is installed.

Disabling Resource Synchronization in Engine Configuration

To change resource synchronization on a per-Engine basis, in the Engine Configuration, under the **Resource Validation**heading, change the value of **Synchronize Resources**to `False`. When an Engine with this setting disables log in, it does not synchronize resources.

This is useful when you want to isolate a set of Engines and prevent them from doing any resource synchronization from any Broker. This must not be used if you want to deploy your resources manually.

> **Note:**
>
> When resource synchronization is disabled in the Engine Configuration, Engines do not auto-upgrade themselves when an update is installed or when the grid is upgraded.

Bridges

Bridges are Grid Libraries that enable Engines to run non-Java Services, such as C++, .NET, and R. All Bridges are pre-packaged and deployed in the Grid Library replication directory when you install or upgrade the GridServer Manager.

JREs

In the rare event that a particular service cannot use the default JRE that is deployed to the Engines, a JRE can be packaged as a Grid Library. The Service's top-level Grid Library then declares it as a dependency. When an Engine takes a Task, it then restarts using this JRE. Note that the JRE must be a supported version.

JREs are packaged as `jre_name-version.gz` or `jre_name-version.zip` where, `jre_name` includes `jre-os`. The `version` is the JRE version, for example, `11.x.x`. The os is the platform, such as `linux64`, `win64`, `linux`, or `win32`.

For example, for linux 64: `jre-linux64-11.x.x.tar.gz`.

For a JRE Grid Library, you can optionally specify JVM arguments in the XML. To do so, add an `<arguments>` element to the root element. It can take any number of `<property>` elements, each containing a `<name>` element and an optional `<value>` element.

If the property has a value, the argument `name=value` is added. Otherwise, only the `name` argument is added.

If the same argument is set in the Engine Configuration and the Grid Library, the Grid Library overrides the Engine Configuration.

> **Note:**
>
> Specifying the JVM debug port inside a Grid Library results in unpredictable behavior and is not supported. Set this functionality with the **Debug Start Port**setting on the **Grid Components > Engines > Engine Configurations**page.

Example: Creating a Multi-Platform JRE Grid Library

Here is an example of how to use the same JDK version Grid Library for Linux and Windows:

> **Note:**GridServer 7.1.1 includes Java Runtime Environment (JRE) gridlibs for Java 11, 17, and 21, but does not include a gridlib for Java 11. Note that Java 17 and 21 do not include separate JREs in the GridServer 7.1.1 package.

1. Download the target JDK version installer for each targeted platform from the Oracle website, https://www.oracle.com/in/java/technologies/javase/jdk11-archive-downloads.html. For example, for Linux64, download `jdk-11.x.x_linux-x64_bin.tar.gz` and for Windows-x64, it is `jdk-11.x.x_windows-x64_bin.zip.`
2. Extract the downloaded JDK `tar.gz` file for Linux or `.zip` file for Windows at the desired location.
3. Create a `grid-library.xml` file with the following contents:

For Linux:

```
<?xml version="1.0" encoding="UTF-8"?>

   <grid-library jre="true" os="linux64">

   <grid-library-name>jre-linux64</grid-library-name>

   <grid-library-version>11.x.x</grid-library-version>

   <lib-path>

      <pathelement>./bin</pathelement>

      <pathelement>./lib</pathelement>

      <pathelement>./lib/server/</pathelement>

   </lib-path>

   </grid-library>
```

For Windows:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library jre="true" os="win64">

<grid-library-name>jre-win64</grid-library-name>

<grid-library-version>11.x.x</grid-library-version>

<lib-path>

   <pathelement>jre/bin</pathelement>

   <pathelement>jre/lib</pathelement>

   <pathelement>jre/lib/server/</pathelement>

</lib-path>
```

```
</grid-library>
```

Note that the order of `pathelement` is important. For example, if you want the server VM, then `jre/lib/i386/server` must come before `jre/lib/i386`. This also applies to the `native_threads` directory.

4. Create the Grid Library archive for each platform containing the `grid-library.xml` that you created above and save the `grid-library.xml` in the extracted JDK folder.
5. Rename the JDK folder to JRE folder for the targeted platform and OS with the JRE version number.For example, `jre-linux64-11.x.x` for Linux and `jre-win64-11.x.x` for Windows.\
6. Archive the JRE folder to create JRE package as `tar.gz` file for Linux and `.zip` file for Windows.For example, `jre-linux64-11.x.x.tar.gz` for Linux and `jre-win64-11.x.x.zip` for Windows.
7. Navigate to **Services > Services > Grid Libraries**.
8. Select **Upload Grid Library**at the top-right of the page.
9. Click **Choose File**, browse to choose the `tar.gz` or `.zip` JRE file, and then click **Upload**.
10. Add the following dependencies to your application:

```
<dependency>
   <grid-library-name>jre-win64</grid-library-name>
   <grid-library-version>11.x.x</grid-library-version>
</dependency>
<dependency>
   <grid-library-name>jre-linux64</grid-library-name>
   <grid-library-version>11.x.x</grid-library-version>
</dependency>
```

After performing these steps your application uses the exact JRE version that you have specified for each platform.

Packaging Grid Libraries

For more information about how to create and package Grid Libraries, see the *GridServer Developer's Guide*.

Distributing Grid Libraries

The GridServer system provides a Resource Deployment mechanism for securely distributing Grid Libraries. The Grid Libraries to be deployed are uploaded to the Primary Director. The resources on the Director are synchronized to Brokers, and then Brokers synchronize the files with Engines. The files are secure in that they cannot be accessed by anyone on the network, only the Engines.

Maker/Checker Support

In an enterprise grid environment, there might be multiple groups responsible for creating resources. It might be necessary to minimize contention between the groups. For instance, two groups might use the same core third party library and have packaged it as a Grid Library; but each might have assigned the same version number to a slightly different patch level, causing Engines to restart frequently.

To address this, there are two sets of resources for two different user roles. The "maker" user has access to a staging area, where they can upload resources to a staging area. The "checker" user then validates the resources (ensuring that there is not a version conflict in our example) and deploys the resources. A "maker" user has a Security Role with access to the Resource Deployment Maker feature (such as the default Manage role.) A "checker" has access to the Resource Deployment Checker feature (such as the default Configure role.)

If a Director to Broker sync fails for any reason, or if a file sign check fails for any reason, the Broker to Engine sync is disabled until the next successful sync or sign. When in this state, Engines act as if sync is disabled, and continue being operational with their current set of resources.

The Resource Deployment Interface

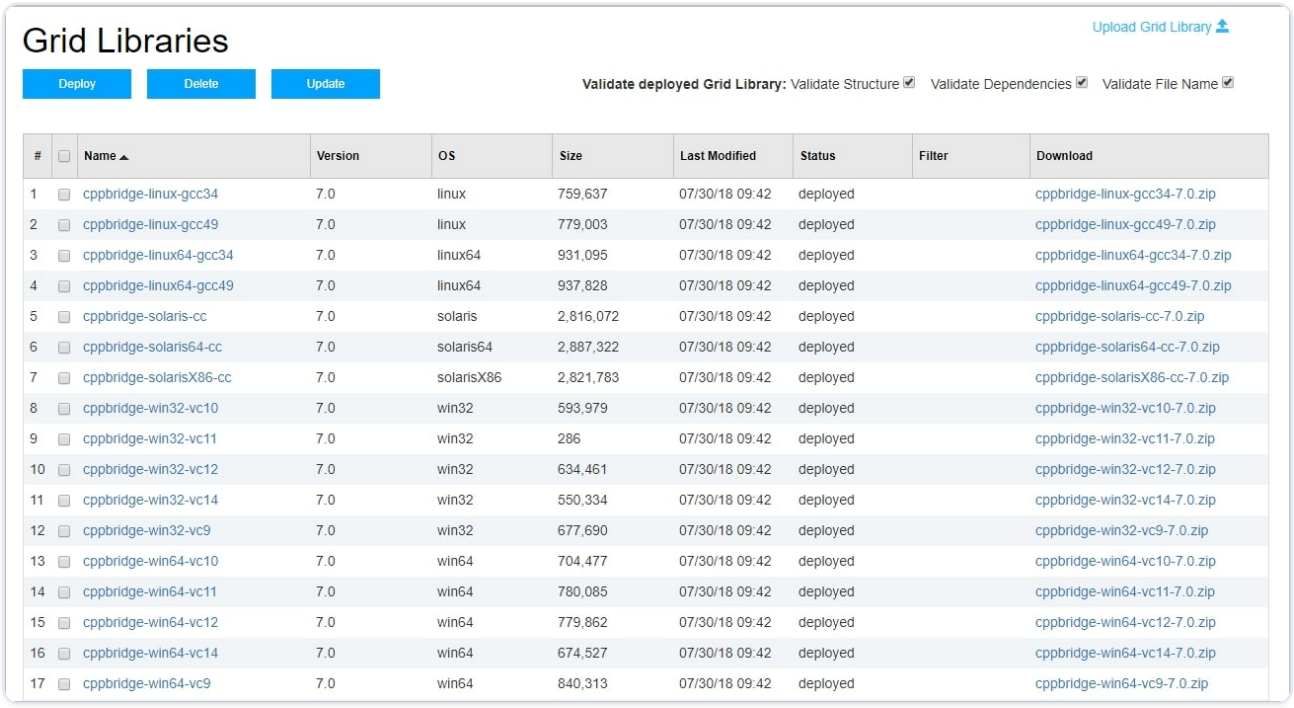
The Administration Tool provides a graphical interface to manage resources synchronized to Engines. To manage resources, on the Primary Director, go to **Services > Services > Grid Libraries**. The Grid Libraries page features a file browser that can be used to manage Grid Libraries.

The **Grid Libraries**page displays a list of Grid Libraries. To upload a Grid Library or Filter:

1. Click **Upload Grid Library**at the top of the list.

2. Click **Choose File**, browse to a file, and then click **Upload**.

   This transfers the file into the staging directory. To download a Grid Library, click the file name in the list.



#	Name	Version	OS	Size	Last Modified	Status	Filter	Download
1	cppbridge-linux-gcc34	7.0	linux	759,637	07/30/18 09:42	deployed		cppbridge-linux-gcc34-7.0.zip
2	cppbridge-linux-gcc49	7.0	linux	779,003	07/30/18 09:42	deployed		cppbridge-linux-gcc49-7.0.zip
3	cppbridge-linux64-gcc34	7.0	linux64	931,095	07/30/18 09:42	deployed		cppbridge-linux64-gcc34-7.0.zip
4	cppbridge-linux64-gcc49	7.0	linux64	937,828	07/30/18 09:42	deployed		cppbridge-linux64-gcc49-7.0.zip
5	cppbridge-solaris-cc	7.0	solaris	2,816,072	07/30/18 09:42	deployed		cppbridge-solaris-cc-7.0.zip
6	cppbridge-solaris64-cc	7.0	solaris64	2,887,322	07/30/18 09:42	deployed		cppbridge-solaris64-cc-7.0.zip
7	cppbridge-solarisX86-cc	7.0	solarisX86	2,821,783	07/30/18 09:42	deployed		cppbridge-solarisX86-cc-7.0.zip
8	cppbridge-win32-vc10	7.0	win32	593,979	07/30/18 09:42	deployed		cppbridge-win32-vc10-7.0.zip
9	cppbridge-win32-vc11	7.0	win32	286	07/30/18 09:42	deployed		cppbridge-win32-vc11-7.0.zip
10	cppbridge-win32-vc12	7.0	win32	634,461	07/30/18 09:42	deployed		cppbridge-win32-vc12-7.0.zip
11	cppbridge-win32-vc14	7.0	win32	550,334	07/30/18 09:42	deployed		cppbridge-win32-vc14-7.0.zip
12	cppbridge-win32-vc9	7.0	win32	677,690	07/30/18 09:42	deployed		cppbridge-win32-vc9-7.0.zip
13	cppbridge-win64-vc10	7.0	win64	704,477	07/30/18 09:42	deployed		cppbridge-win64-vc10-7.0.zip
14	cppbridge-win64-vc11	7.0	win64	780,085	07/30/18 09:42	deployed		cppbridge-win64-vc11-7.0.zip
15	cppbridge-win64-vc12	7.0	win64	779,862	07/30/18 09:42	deployed		cppbridge-win64-vc12-7.0.zip
16	cppbridge-win64-vc14	7.0	win64	674,527	07/30/18 09:42	deployed		cppbridge-win64-vc14-7.0.zip
17	cppbridge-win64-vc9	7.0	win64	840,313	07/30/18 09:42	deployed		cppbridge-win64-vc9-7.0.zip

*The **Services > Services > Grid Libraries**page*

Each Grid Library can be in one of the following states:

- New — An undeployed resource in the staging directory.
- Deployed — A deployed resource in the deployment directory.
- Error — There was an issue with an uploaded file.

  The following conditions cause an error in an uploaded Grid Library:

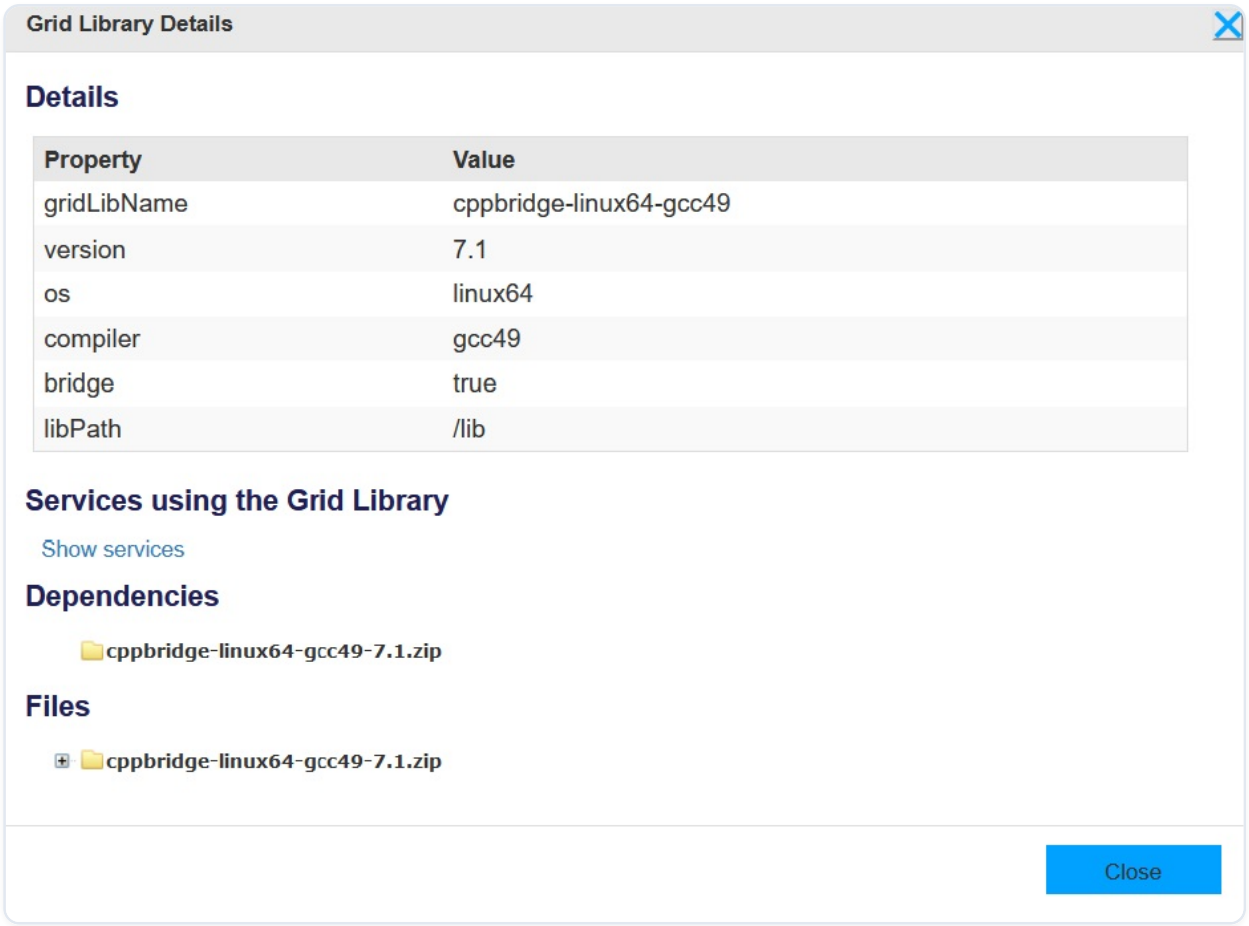
- A Grid Library in the staging directory has the same name as a Grid Library in the deployment directory. Delete a deployed Grid Library before uploading a replacement. The Administration Tool returns an error message if you attempt to upload a file with the same name as a deployed file. If a resource is in error status, the first line displays attributes of the resource file in the deployment directory, and the second line displays attributes in the staging directory.
- The Grid Library's archive file is corrupt.
- The Grid Library's `grid-library.xml` file does not validate against the Grid Library DTD.

  The following buttons can be used on selected Grid Libraries:

- **Deploy**— Deploy selected resources from the staging to the deployment directory.

- **Delete**— Delete selected directories and files from the staging and deployment directory. If the resource is in error status, only its staging copy is deleted.
- **Update**— Force an immediate file signing request and populate changes to the Engines.

You can also show details for a Grid Library by clicking its name in the list. This opens a Grid Library Details window, which displays the following:



The Grid Library Details page

- Properties and values set in the Grid Library's `grid-library.xml` file.
- All Services using the Grid Library, directly or indirectly.
- A list of the Grid Library's dependencies. If there are multiple dependencies, this is a tree that you can browse and can click to view all dependencies.
- A tree of all files and directories in the Grid Library that you can browse.

Grid-Specific Resources for Multiple Grids

If you migrate Engines among multiple grids, you can configure GridServer to use a different resources directory for each grid. This eliminates full Grid Library synchronizations when Engines are migrated among grids, such as when using Federator to manage these grids.

To enable this feature, go to **Admin > System Admin > Manager Configuration > Resource Deployment**and in the Director settings, change the value of **Use Grid Specific Resources**to true. If set to true, resources are downloaded to `resources/grid-`*gridID* in the Engine installation directory, where *gridID* is the Manager ID of the Primary Director instead of the `resources` directory. (For example, on a Windows Engine, this can be `C:\TIBCO\DataSynapse\Engine\resources\grid-1925105476`.) Note that the Primary Director must be restarted for the change to take effect.

You can also configure how many grids Engines can synchronize with by changing the value of **Number of Grids to Sync**. This value specifies the number of grids that have resources retained on the Engine. If the value is set to N and an Engine already has resources that are downloaded from N+1 grids, the resources from the least recently visited grid are deleted to conserve disk space. This value is a Director-level setting and must be set consistently across multiple grids to reduce resource re-download effectively. Note that all Brokers must be restarted for the change to take effect.

Grid Library Filters

You can limit which Engines synchronize a Grid Library by using Grid Library filters. This enables you to define a filter file in XML that specifies the Engines based on Engine properties. This decreases hard drive usage and network bandwidth in deployments where not all Engines can or are allowed to work on certain types of Services.

A Grid Library filter is an XML file with the same name as its associated Grid Library and ends with the extension `.filter`. When an Engine requests a resources sync, the list of Grid Libraries are filtered based on the matching of the conditions.

Grid Library Filter Document Type Definition (DTD)

A Grid Library filter has the following DTD:

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
`grid-library-filter`	root element	Elements • `property-condition-set*`

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
`property-condition-set*`	A set of conditions that must be satisfied for the Engine to synchronize	Elements • `property-condition*` • `property-condition-set*` Attributes • `operator?`
`property-condition*`	A condition that must be satisfied for the Engine to synchronize	Elements • `property` • `comparison` Attributes • `null-compare?`

A `property-condition` uses the following comparison attributes:

- `contains`
- `notContains`
- `matches`
- `notMatches`
- `equal`
- `notEqual`
- `greaterThan`
- `lessThan`
- `greaterThanEqual`
- `lessThanEqual`

The `operator` attribute can be `and` (the default) or `or`. The `null-compare` attribute can be `false` (the default) or `true`.

Filter Example

The following code is an example of a simple filter which only deploys a given Grid Library to Engines running four or more instances:

```
<grid-library-filter>
  <property-condition-set>
    <property-condition>
      <property>
        <name>numInstances</name>
        <value>4</value>
      </property>
      <comparison>greaterThanEqual</comparison>
    </property-condition>
  </property-condition-set>
</grid-library-filter>
```

Deploying Grid Library Filters

Grid Library filters are uploaded in the GridServer Administration Tool at **Services > Services > Grid Libraries**, using the same control used to upload Grid Libraries. When you upload a filter, it is first validated to ensure it is valid XML and the comparison, property, value, and type elements are valid and `property.name` elements are not empty. Also, there is a 16 KB size limit for filters, which is checked.

After the filter is validated, you are prompted if you want to deploy it. If you click **Cancel**, the filter is not added to the Manager.

Deployed filters are shown in the list of entries on the **Services > Services > Grid Libraries**page, and you can delete them the same way you delete a Grid Library. Unlike Grid Libraries, you can upload a filter with a duplicate name to replace an existing filter with a revised version.

JAR Ordering File

If you are using multiple JAR files and need the classloader to load them in a specific order to prevent conflicts, you can specify the order in which they are loaded. To do this, create a file called `index.libs` in the JAR path root and put the names of the JAR files, one per line, in the order in which they must be loaded. Those not in the list load afterward in no specific order.

Uploading and Deploying with the Admin API

You can programmatically upload, deploy, download, and delete Grid Libraries and filters to a Manager using the JDriver Admin API. The `ServiceAdmin` class has methods you can use to manipulate resources on a Manager. It includes the following related methods:

- `uploadResource` – uploads a resource into the temporary staging directory
- `deployResources` – copies resources from staging to deployed
- `downloadResource` – downloads a copy of a resource
- `deleteResources` – removes resources
- `listResources` – returns a list of resources on the Manager

You can also determine Grid Libraries loaded on Engines by using the `EngineAdmin` class to get the `EngineProperties`. The `GRID_LIBRARIES` property is a list of all downloaded Grid Libraries, and `GRID_LIBRARIES_UPDATE_TIME`, is the time at which they were last updated.

For more information, see the Java API documentation.

Running Services

To run a Service, you must deploy the Service's classes to Engines, and set options that are used when the Service is used. This is done in the Service Type, which specifies the Grid Library containing a Service's implementation, binds classes to Service methods, and sets any Service options. One of the options that can be set is Service Run-As, or the user account that is used to run the Service.

Registering a Service Type

To use a Service, you must first register a Service Type from the GridServer Administration Tool.

To register a Service Type:

1. Go to **Services > Services > Service Types**.

2. A list of existing Service Types appears on that page, along with a line for adding a new Service Type.
3. Enter the Service Type Name on the blank line.
4. Select the Service Implementation, then click **Add**.
5. A window with several options appears after clicking the **Add**button.
    - For Java Service Types, enter the fully qualified class name for the Service.
    - For .NET Service Types, enter the class, assembly, and domain name for the Service.
    - For R Service Types, enter the function interface for the Service.
    - For dynamic libraries, enter the library name for the Service.
    - For commands, enter the command line for the Service.

    The window also allows you to enter options for the Service Type.

> **Note:**After you register a Service Type, you must deploy the implementation to your Engines. If you submit tasks without deploying a Service implementation, the Driver submits the tasks with no error, but no Engines take tasks.

Service Run-As

There are often cases where Services require specific user permissions to access needed resources. By creating the Engine process as a given user, all Service invocations that are run by the Engine can operate with these permissions. **Service Run-As**(or **RA**) allows for specification of authentication domain accounts under which Service invocations run.

By default, all RA credentials authenticate on the Engine Daemon to verify that the credentials are valid for the Engine's authentication domain. You can disable Service RA authentication on the Broker, but do so only when you have a specific reason. For example, if you are using Kerberos or Windows authentication, you must disable this if your Drivers have Negotiate enabled, since there is no password available.

compressData

If you want the initial input and output data to be compressed, you must set the **compressData**field as `True`. To disable the feature, set the value of the **compressData**field as `False`.

Default value: Not set

encryptionEnabled

If you want the initial input and output data to be encrypted, you must set the **encryptionEnabled**field as `True`. To disable the feature, set the value of the **encryptionEnabled**field as `False`.

Default value: Not set

Types of Credentials

You can specify Service Run-As credentials for a given Service in one of two ways: as stored credentials or pass through credentials.

- **Stored Credentials** — Enter Service Run-As credentials on the Director with the GridServer Administration Tool. These credentials are synchronized with all Brokers. These credentials are linked to Services in the Service Type Registry by specifying the username in the **RunAsUser**field. Credentials in the repository consist of a username and a password. The username can be in Windows `DOMAIN/username` format if domain-specific authentication is required. UNIX Engines ignore this domain.
- **"Pass through" Credentials** — The Driver provides the username of the current Principal that is logged in and is running the Driver. The password is provided as a DriverManager property, `CURRENT_USER_PASSWORD`. These are referred to as "pass through" credentials. A password set on the Driver is necessary to prevent user account spoofing between authentication domains. For example, logging in as a local user on the Driver machine to pose as an LDAP user in the credentials DB.

Pass through credentials are indicated for a Service in the Service Type Registry with the `$` token. This token is substituted with the username of the current principal that is running the Driver process. The token might also be prefixed with a Windows domain if domain-specific authentication is required. UNIX Engines ignore this domain.

> **Note:**
>
> When you create a Service using pass through credentials and the `Collection.NEVER` Service option, failover does not work under all circumstances.
>
> - If the Broker accepts the Service and the Engine has not yet picked up the tasks, failover works correctly and tasks rerun successfully.

- If the Broker accepts the Service and the Engine has already picked up the tasks, tasks do not rerun successfully, and an Exception is generated. Pass through credentials do not persist through failover.

Using Run-As

To use Run-As, you must do three things: set up Engines, add credentials, and associate credentials with Service Types.

Engine Setup

For information about how to set up Engines for Service run-as, see “Configuring Run-As for Windows Engines” and “Configuring Run-As for UNIX Engines” in the *GridServer Installation Guide*.

Managing Credentials

The Credentials DB is a store of RA credentials on the Director and Brokers used for RA services. It is maintained on the Director and synchronized with Brokers.

The **Run-As Credentials**page in the Administration Tool lets you create, edit, and delete RA credentials.

To add new Credentials to your Manager:

1. In the Administration Tool, go to **Admin > User Admin > Run-As Credentials**.
2. Enter the name of a credential, a password, and then enter the same password again.
3. Click **Add**.

Managing Run-As in Service Types

The Service Type Registry entries allow specification of an RA user name for use with that Service. To specify a Run-As user for a Service Type:

1. In the GridServer Administration Tool, go to **Services > Services > Service Types**.
2. Select an existing Service Type, click to the **Actions**control, and select **Edit Service Type**. This opens the Service Type Editor window.
3. In the Service Type Editor window, under the **ContainerBinding**header, enter the user name in **RunAsUser**.

Note that in this field, you can use `$` to indicate the Driver's current user. Leaving this value blank (the default) indicates that the process runs as the same user running the Engine Daemon.

If you are adding a user name that contains unicode characters, you must change to the correct code page to match the user name.

The **Specify Additional RunAs User**permission is needed to specify a user other than the Driver or Engine Daemon process.

It is also possible to specify a Windows domain in the **RunAsUser**field. For example, if you are using a UNIX Driver (which is not in a Windows domain) and you want to run Services on Windows Engines using a specific user and domain, you can specify this in the form `domain/username`. The forward slash translates to a backslash. For example, specifying `DATASYNAPSE/BILL` runs Services as the user `BILL` in the `DATASYNAPSE` Windows domain (`DATASYNAPSE\BILL`).

Scheduling

One of the responsibilities of Brokers is *scheduling*, which is the management of Services and tasks on Engines and interactions between Engines and Drivers. This section gives more details on how scheduling works, and the method used to determine what tasks in a Service are sent to what Engines.

Most of the time, the scheduling of Services and tasks on Engines is completely transparent and requires no administration. However, to tune performance, or to diagnose and resolve problems, it is helpful to have a basic understanding of how the Broker manages scheduling.

Recall that clients create Service Sessions on the Broker. Each Service Session consists of one or more tasks, which might be performed in any order. The scheduler determines the optimal match of Engines to Services. Whenever an Engine reports to the Broker to request work, the Broker assigns a task from that Service to the Engine. When an Engine completes a task, it is queued on the Broker for collection by the client. If an Engine is interrupted during processing, the task is requeued by the Broker.

Reschedules and Retries

Before the discussion of scheduling behavior, we must first define the terms *Retry* and *Reschedule* within the context of scheduling tasks.

Retry

A Retry is when a task is re-queued due to a known failure of the task. Such failures can be due to an error condition in the implementation, an error due to inability to download data, or the failure of

an Engine (the monitor has detected that the Engine is no longer connected but it has not logged off.) It is always the result of the Engine returning the task as failed to the Broker. When a task is retried, it is always placed at the front of that session's queue. The scheduler manages a retry count for each task, so that a limit can be placed on the number of allowed retries.

Reschedule

A Reschedule is when a task is re-queued when it might or might not have failed. When a task is rescheduled, it is by default placed at the back of that session's queue, unless the **Reschedule First**configuration option on the Broker is set to true. (Go to **Admin > System Admin > Manager Configuration > Services**to set it.) The scheduler also manages a reschedule count for each task. The following conditions result in a reschedule:

- Engine Logoff: When an Engine logs off gracefully while running a task (such as when UI or CPU idle conditions are met, or there is a forced rebalance), the task is rescheduled, but the reschedule count is not incremented, since there was no task error.
- Redundant Rescheduler: If any of the Redundant Rescheduler strategies are in effect, tasks might be rescheduled to other Engines. By default, those tasks are allowed to continue to run on the current Engines, in case they finish before the rescheduled tasks. In this case, the reschedule count is increased.

Timeout Behavior

When the `INVOCATION_MAX_TIME` option is set, it specifies that any invocation of a request might not exceed this value. If a task times out on an Engine, it can either retry or be rescheduled, depending on what makes more sense for your application. If retried, the current Engine's invoke process ends, and the task is assigned to another Engine. If rescheduled, the current Engine task continues execution. In either case, the appropriate count is incremented.

The default behavior (retry) is set on the Broker. It can also be set for the Service Type on the Service Type Registry page, or programmatically when the Service Session is created.

The specific timing involved with a retry/reschedule depends mainly upon three properties: The Task Max Time, the scheduler interval and the Engine heartbeat.

The moment the task is picked up by the Engine, the start time is marked. The scheduler wakes up at least once in every scheduler interval seconds to check for any tasks in progress that exceed the max time. If **Reschedule on Timeout**is false (the default), the scheduler logs off Engines that have timed out, causing the tasks to retry immediately. The tasks are placed at the top of its session's queue. Note that the Engine on which the task is running does not restart itself until the next message is sent, typically a heartbeat. If true, those tasks are redundantly rescheduled, and

Engines that have timed out are allowed to continue; the task is complete as soon as any Engine completes it.

In general, the maximum time is the Task Max Time plus the Poll Period of Service Rescheduler. For instance, if Task Max Time = 50 sec and Poll Period= 60 sec, best = 50 secs and worst = 110 secs. However, it can take up to the Engine Heartbeat for the Engine on which a task was retried to log back in.

The Scheduler

The scheduler is the component that is used on a GridServer Broker to assign tasks to Engines. It attempts to make optimal matches based on criteria such as the session priority level or SLA group, affinity, and Serial Service and Priority execution modes.

Scheduler Overview

The scheduler aims to schedule tasks to Engines by attempting to have the proper amount of Engines allocated to all active Service Sessions at any given time. A *Scheduling Event* is any event that might result in a task being assigned to an Engine, such as an Engine finishing a task, an Engine logging in, or new tasks added. On any given scheduling event, the scheduler decides the number of Engines each Session must have at the time, based on static and dynamic criteria, and then assigns the appropriate number of Engines to sessions based on how many the Session needs to reach the ideal level.

There are two modes under which the Scheduler can operate: **Priority**, or **Service Level Agreement (SLA)**.

Priority Mode

Every GridServer Service has an associated **priority**. By default, there are ten priority levels, ascending in priority 0-9. When set to 0, the Service is suspended, meaning that no tasks are assigned. Priority can also be set to **Urgent**, which is covered later in this section. The priority is set when a Service is created. It can be changed at run time in the Administration tool and by the Admin API.

A **Priority Weight**is associated with each Priority Level. The weight defines the amount of Engines allocated to a session relative to all other active sessions. For example, if Session A and B have weights of 2.0, and Session C has weight 4.0, and there are eight Engines, Session A and B get allocated two Engines each, and Session C gets four. To set the weights, go to **Admin > System Admin > Manager Configuration > Services**and change the **Priority Weights**property. By default the weights are linear.

The number of priority levels can be changed. However, a large number of priority levels can impact performance when Serial Priority is not enabled, so it is recommended that the Serial Priority is enabled in this case.

There are two algorithms that are used in Priority mode, **Usage**, which allocates Engines to all running Services as fairly as possible, and **Time**, which simply allocates Engines to Sessions in the order in which they were created. Also, when **Serial Priority Mode**is enabled, Sessions of a higher priority are assigned Engines when needed before lower priority Sessions. By default, Usage is used with Serial Priority Mode disabled.

Usage Algorithm

The scheduler takes into account the amount of usage that the Session has received over a given historical window of time. The “usage” refers to the amount of Engine clock time that the Session has occupied during that window. When a Session is created, it is initialized in such a way that it simulates as if it was running ideally over this window.

This usage provides the ordering in which Engines are allocated to Sessions. This addresses starvation issues, round-off error (the number of ideal Engines is rarely an integer), and under/over-utilization due to discrimination, changes in the number of available Engines, and so on.

On a scheduling event, Sessions are assigned the ideal number of Engines less the amount that are currently allocated, in the order of least to most usage.

This approach can be seen as analogous to a CPU thread scheduling algorithm. Each Session is a “thread”, the Engines are the “CPU”, the window is the sample period, and each task is an uninterruptible unit of CPU time allotted to a thread.

Whenever an Engine or set of Engines is available for scheduling, the scheduler decides how many Engines each session must be allocated. In general, that value is:

Ideal Engines per Session = All Engines * Session Priority Weight / Total Weight,

where “Total Weight” is the sum of all Priority Weights of active sessions. This value is rounded up to the next integer to prevent starvation for an ideal calculation of < 0.5, and assures that the sum of Ideal Engine’s is always at least as large as Total Engines. This algorithm also takes into account if the actual number of Engines that can be allocated is less than the ideal, such as when a Session is towards the end, or when Max Engines is used.

Recall that a Session’s usage is considered to be the total Engine clock time spent on the session over the last configurable amount of time. This includes running and completed tasks. When a Session is created, it must initialize its usage. The simplest, most fair method of doing this is to

assume it has been operating in a steady state over the window with the ideal non-rounded number of Engines. The variables that monitor usage are then initialized as such. If no sessions are active, it initializes them such that the session's ideal is the total number of Engines currently on the Broker.

Whenever there is an event that requires a scheduling episode, the scheduler assigns the proper number of Engines to each session for it to be at its ideal amount. This assignment is performed in order of least to most priority-normalized usage. If there are any unassigned Engines remaining after this initial round based on usage (typically due to disallowed conditions preventing assignment), a second tier round robin assignment is performed.

Time Algorithm

The Time algorithm is used by setting Serial Service Execution to true. This algorithm works as follows:

When a Session is created on the Broker, it is placed in the queue. On each scheduling episode, the scheduler simply iterates through the queue and assigns all idle Engines it can to each Session. Normally, only the first Session is assigned Engines, except when that Session is finishing up, or if Discriminators prevent Engines from running on that Session. A Session keeps its place in queue until it is destroyed regardless of whether or not it has tasks in queue or running.

Serial Priority Mode

When Serial Priority Mode is enabled, the scheduler ensures that Sessions are assigned Engines in order of priority. The scheduler iterates through each priority level in descending order, and assigns as many Engines to Sessions at that level as possible. Either the Time Algorithm or the Usage Algorithm, depending on whether Serial Service Execution is enabled, is used on the subset of Sessions at the same Priority level. Note that this means that Priority takes precedence over creation time.

Intrinsic Affinity

The scheduler uses the fact that an Engine has initialization data and updates from a particular Service to prioritize routing of subsequent requests to that Service. This feature, called *affinity*, reduces data movement, because unneeded Engines are not recruited into the Service. For example, Engine A has worked on Session X and Engine B has not. If both are idle and a task is submitted by X, Engine A is assigned the task. However, if Engine A is busy, Engine B is assigned the task. You can also use the `AFFINITY_WAIT` Service option to control how long a queued request avoids allocation to an available Engine that has no affinity, in the hope of later being matched to an Engine with affinity.

Affinity is not used when using the Time algorithm.

For more information about tuning or customizing how the scheduler uses affinity, see Optimizing the Grid.

Priority Aggregation

Priority Aggregation is a setting that can be enabled for the usage algorithm. When enabled, the amount of Engines to be allocated is now aggregated over the entire group of Sessions running at a priority level, rather than per Session. That is,

Ideal Engines per Session = All Engines * Session Priority Weight / Total Weight / Sessions at Priority

This mode is used when you want to guarantee a known distribution of Engines amongst priority levels regardless of how many Sessions are running at that level.

Example:

With 100 Engines total, 1 Session at level 6 gets 60, and 1 Session at level 4 gets 40.

Without priority aggregation, if another level 4 Session is added, each level 4 Session now wants 29, and the level 6 wants 43. With it enabled, the level 6 Session still gets 60, and each level 4 Session gets 20.

Urgent Priority Services and Preemption

A Session's priority can be set to **Urgent**when that Session must be serviced immediately, even preempting running tasks if necessary. An urgent Service's weight is hard-coded to be essentially infinite, so that they are assigned all available Engines.

When an Engine is preempted, the task it is currently running is canceled and rescheduled, and the Engine becomes available for new tasks. Engines are preempted on a Service under the following conditions: if after being assigned all free Engines a Service can still make use of more Engines, then it might preempt some busy Engines, subject to two constraints that can be adjusted with configuration properties. First, the urgent Service must have been in the queue for **Preempt Delay**Seconds. Second, the percentage of Engines in the grid running urgent Services cannot exceed **Preemptable Engine Percent**.

For example, if this property is set to 50, and 47 percent of the Engines are currently running urgent Services, then at most three percent are preempted. This value is not a hard limit on the

number of Engines that might be running urgent Services, because free Engines are allocated to urgent Services regardless of how many Engines are already running urgent Services.

The scheduler chooses Engines for preemption based on the following rules: Engines running an urgent Service are never preempted. An Engine running a task from a Service with lower priority is generally selected in preference to one running a higher-priority task. However, if the lower-priority task has been running for a long time, a short-running, higher-priority task might be preempted instead. The **Preempt Threshold Minutes**property determines the value at which this crossover happens. For example, if this property is set to 30, then an Engine that has just started running a priority 2 task is chosen for preemption over an Engine that has been running a priority 1 task for more than 30 minutes. The formula is as follows: `priority + (runningMillis / preemptThresholdMillis)`.

> **Note:**
>
> Preemption can have a significant performance impact on your grid and cause scheduling problems with other Services. It must be used with caution.
>
> Other important points concerning priority Services and preemption:

- Tasks canceled by preemption are not subject to a rescheduling limit, since they are not considered failures.
- To prevent preemption from ever occurring, set **Preemptable Engine Percent**to 0.
- The first Service on the queue might not get all free Engines if it doesn't have enough tasks, it is already using its maximum number of Engines, or it discriminates against some Engines. Free Engines that are not taken by the first urgent Service are first offered to the other urgent Services on the queue, and then to all other Services.

SLA Mode

When the SLA mode is used, the scheduler guarantees that a number or percentage of Engines on the Broker is allocated to a group of Services, provided enough Engines are available.

To enable SLA scheduling, go to **Admin > System Admin > Manager Configuration > Services**and set **SLA Scheduler Enabled**to true. Then you must define the Broker's **SLA Groups**, which is a comma-delimited list of groups and values. The values must be either all

integers, or all floating points where 0 < x <1. When integers, it indicates that the SLA is the actual number of Engines, otherwise it is the percentage of Engines currently logged in to the Broker.

For example, setting SLA Groups to `a=10,b=12,c=20` specifies that group A's target is 10 Engines, group B's target is 12 Engines, and group C has a 20-Engine target. Alternatively, an SLA Groups setting of `a=.25,b=.5,c=.1` sets the A, B, and C's targets to 25%, 50%, and 10% of Engines, respectively.

Sessions are assigned to groups by setting `Description.SLA_GROUP_NAME` on the Description. All Sessions must have this set; if not set or set to an invalid group name, the Session is rejected by the Broker.

On every scheduling event, the scheduler calculates how many Engines must be allocated to each SLA Group, and then assign Engines to Sessions as follows:

- The entire list of Sessions is ordered by the same **Usage**algorithm used in the Priority mode. This ensures fairness in the steady-state. Scheduling is performed round-robin within an episode, starting at the Session with the least amount of usage.
- When a Broker has enough Engines to meet all group SLA Engine needs, Engines are first assigned so that all SLA Groups have their SLA number met. Then any remaining Engines are divided up among the groups, weighted by their SLA numbers, rounded to the next integer.
- When a Broker does not have enough Engines to meet group SLA Engine needs, the Engines are allocated to groups by weight as in the case of the remainder when there are enough Engines.

For example, you have set **SLA Groups**as `a=20, b=30, c=50`, and there are nine remaining Engines after SLA Engine needs are met. 9 * 20 / 100 = 1.8, so group A gets 2 remainders. Likewise, group B gets 3, and group C gets 5. Note that in this case, one of the groups gets one less than their number due to rounding, but the usage algorithm corrects these inequities in the long run.

The SLA scheduler does not take Affinity into account, nor is there any analog to Urgent Priority.

SLA Task Preemption

In the case of long-running tasks, you can use the task preemption option with the SLA scheduler to better distribute Services to idle Engines. When the **SLA Preemption Enabled**property is set to true, when one SLA group is idle, other groups might be allocated its Engines. If Services are then added to the idle group, tasks are preempted to reallocate Engines needed to meet its SLA.

Common Scheduler Features

The features in this section are common to both scheduler modes.

Grid Library Aware Scheduling

The GridServer scheduler does not schedule a task to an Engine if that Engine does not yet have the root Grid Library for that Service. Additionally, when an Engine logs in, it does not wait until it is synchronized to run tasks. Rather, it works on any Services it can while it is synchronizing any new libraries. This allows for straightforward library deployment on large grids where it might take hours to fully sync. Services can be started at any time instead of waiting until all Engines are synced.

Engine Blacklisting

If a Service sets the option `engineBlacklisting` (`ENGINE_BLACKLISTING`) to true, then Engines that fail on a task from that Service do not receive any other tasks from that Service. The default is false. '"fail" means any action that results in a failed task being sent back to the Manager, regardless of whether that failure was due to Engine hardware, Engine environment, or Service implementation code. It does not include events such as the Engine going offline to user activity, since that does not result in a task failure.

You can also set the option `failuresBeforeBlacklist(FAILURES_BEFORE_BLACKLIST)` to a number of task failures before an Engine is blacklisted.

Blacklisted Engines are excluded for a particular Service Session only. They can freely accept tasks from any other Service, regardless of Service Type, assuming the other Services haven't also blacklisted the Engine or have some Conditions in place that prevent it. Blacklisted Engines can also be shared with other Brokers that need Engines.

To remove an Engine from all blacklists, go to **Grid Components > Engines > Daemon Admin**and select **Clear from Blacklists**from the **Actions**list.

You can get a list of blacklisted Engines using the GridServer API. In Java, the `getBlackListedEngines` method in `com.datasynapse.gridserver.admin.ServiceAdmin` retrieves a list of Engines that have been blacklisted for a given Service. It is also available for C++ and .NET. See the GridServer API for more information.

Engine Greylisting

If your Tasks are more heterogeneous, you might not want to use blacklisting, because a single Task failure might not imply failure of all Tasks. Greylisting enables you to make it less likely that an

Engine works on a Service Session than other Engines, without completely excluding that Engine. This is done by lowering affinity for an Engine for the Task retry.

To use Engine Greylisting, set the Service option `engineGreylisting` (`ENGINE_GREYLISTING`) to true. When a Task fails, the Service Affinity for that Engine is reduced by a configurable amount, which by default is 5. To change this amount, go to **Admin > System Admin > Manager Configuration**, and under the **Affinity**heading, change **Greylist Affinity**to a negative value.

Greylisting can be used with blacklisting, if the `failuresBeforeBlacklist` is greater than zero. The Service Affinity is reduced upon each Task failure until `failuresBeforeBlacklist` is reached and the Engine is blacklisted.

The **Clear from Blacklists**action described above also clears greylists.

Engine Properties

Engines have a number of intrinsic properties, such as available memory or disk space, performance (megaflops), operating system, and so forth, that the condition can use to define eligibility. Custom properties can also be defined on the grid, and property values can be assigned on a per-Engine basis. These properties are used in a number of the following features.

Engine Tiers

Often time customers might have distinct sets of resources that they might want to use on a preferred basis. Engine tiers provide a mechanism to specify the order in which groups of Engines must be scheduled.

For example, you might have a set of high-performance dedicated blades, a pool of older servers, and a group of desktop computers used as part-time grid resources when idle. Ideally, you might not want to use older servers unless the high-performance servers are completely in use, and avoid using desktop machines unless both other groups were busy.

Engine tiers are defined on the Broker at **Admin > System Admin > Manager Configuration > Services**, under the **Scheduling**heading, with the **Engine Tiers**property. The property's format is an ordered, comma-delimited list of Engine property name-value pairs. For example, for the above scenario, if you had a `type` Engine property, you might use `type=blade,type=server,type=desktop`. The list is ordered from highest-tier to lowest-tier. Engines matching the first tier are always scheduled before those in the second tier, and so on.

If an Engine matches more than one property, the highest tier is matched. If it matches none of the defined tiers, it is scheduled after all other tiers.

Conditions

Conditions are rules that are applied to Services and tasks that affect how they are scheduled to Engines.

A *Discriminator Condition* limits the execution of tasks to a subset of Engines. If an Engine is ineligible to take the next waiting task, it is assigned the first task it is eligible to take. An *Affinity Condition*, like intrinsic affinity, provides for prioritized routing of tasks to Engines, but does not prevent any Engines from taking tasks. These conditions typically are based on Engine Properties. Users can also implement custom versions of these conditions. (Note that Affinity is only used in Priority Mode with the Usage algorithm.)

Task Affinity provides the ability to run a set of Tasks on the same Engine or set of Engines. For example, if a Task loads a large dataset, and a number of subsequent Tasks use the same dataset, you can add a Task Affinity Condition to those Tasks so they prefer to run on the Engine that ran the first Task.

When a *QueueJump Condition* is added to a task submission, the task is added to the front of the Session's queue so that it is the next task taken.

Dependency Conditions are used to create workflow amongst Sessions and tasks. A task can be set to wait until another task or Session completes; likewise with Sessions. Dependent tasks or Sessions can also optionally fail if the dependency fails.

For more information about using Conditions, see the *GridServer Developer's Guide*.

Redundant Rescheduling

Redundant rescheduling addresses the situation in which a handful of tasks, running on less-capable processors, might significantly delay or prevent Service completion. The basic idea is to launch redundant instances of long-running tasks. The Broker accepts the first result to return. Remaining instances are not immediately canceled; it waits to either finish, or waits until the Service finishes. Redundant rescheduling is also useful when completion of long running tasks is critical.

By default, redundant task rescheduling is not enabled. With pools of more capable or nearly identical Engines, fastest task execution occurs when there is no redundancy from rescheduling. In general, rescheduling is only appropriate when there are widely different capabilities in Engines. To enable redundant rescheduling, you must enable one of the three strategies, and set the `REDUNDANT_RESCHEDULING_ENABLED` Service option to true on each Service you want to redundantly reschedule.

Note:

In situations where a group of Engines might slow down a running task, using discrimination can be more efficient than redundant rescheduling.

Three separate strategies, running in parallel, govern rescheduling. Tasks are rescheduled whenever one or more of the three corresponding criteria are satisfied. However, none of the rescheduling strategies apply for any Service until a certain percentage of tasks within that Service have completed. The `Strategy Effective Percent` property determines this percentage.

The rescheduler scans the pending task list for each Service at regular intervals, as determined by the `Poll Period` property. Each Service has an associated `taskMaxTime`, after which tasks within that Service are rescheduled. When the strategies are active (based on the `Strategy Effective Percent`), the Broker tracks the `mean` and `standard deviation` of the (clock) times consumed by each completed task within the Service. Each of the three strategies uses one or both of these statistics to define a strategy-specific `time limit` for rescheduling tasks.

Each time the rescheduler scans the pending list, it checks the elapsed computation time for each pending task. Initially, rescheduling is driven solely by the `taskMaxTime` for the Service; after enough tasks complete, and the strategies are active, the rescheduler also compares the elapsed time for each pending task against the three strategy-specific limits. If any of the limits is exceeded, it adds a redundant instance of the task to the waiting list. (The Broker resets the elapsed time for that task when it gives the redundant instance to an Engine.)

The `Reschedule First` flag determines whether the redundant task instance is placed at the front or the back of the waiting list. That is, if `Reschedule First` is `true`, rescheduled tasks are placed at the front of the queue to be distributed before other tasks that are waiting. The default setting is `false`, which results in less aggressive rescheduling.

Each of the three strategies computes its corresponding limit as follows:

- The `Percent Completed Strategy` waits until the Service nears completion (as determined by the `Remaining Task Percent` setting), after which it begins rescheduling pending tasks that are taking longer than the average completion time for tasks within the Service.

- The `Average Strategy` returns the product of the mean completion time and the `Average Limit` property. That is, this strategy reschedules tasks when their elapsed time exceeds some multiple (as determined by the **Average Limit**) of the mean completion time.
- The `Standard Dev Strategy` returns the mean plus the product of the `Standard Dev Limit` property and the standard deviation of the completion times. That is, this strategy reschedules tasks when their elapsed time exceeds the mean by some multiple (as determined by the `Standard Dev Limit`) of the standard deviation.

Managing Engines

This section contains information for managing GridServer Engines. For information about installing Engines, see the *GridServer Installation Guide*. For information about troubleshooting Engine issues, see Diagnosing Engine Issues.

Engine Routing and Balancing

Engines are dynamically allocated resources. They can migrate among Brokers based on criteria such as load and policy. Use the Engine Balancer to manage log ins and re-route Engines to maintain an optimal balance across the grid. The Engine Balancer is a component of the Director. The Primary Director's balancer always runs. The Secondary Director's balancer runs only if the Primary is down.

The Director handles routing and load balancing as follows:

1. The Director regularly polls Brokers for the states of Engines on the Brokers. The Director tests routing mechanisms against each Engine and determines the optimal location for each Engine. Changes in the states of Engines due to load balancing requirements result in changes in the optimal distribution of Engines on Brokers.
2. The Director sends a request to each Broker that has Engines that must be moved, to log those Engines off.
3. Engines that must return to their home Broker log out immediately, regardless of the task timeout setting.
4. Shared Engines that are busy restart immediately without finishing the current task. Engines that are not busy log out immediately.
5. After an Engine logs off or restarts, it then logs in to the optimal Broker.

Two balancer algorithms are available. Choose one according to how you plan to use the grid:

- The `weight-based` balancer algorithm attempts to distribute Engines equally by relative weights, and it also allows rule-based routing using Engine properties.
- The `Home/Shared Balancer` routes Engines based on an Engine's assigned Home Brokers, and the sharing policy of Home Brokers to other Brokers. Both balancers take into account the number of running and pending tasks on each Broker, and the desired maximum and minimum number of Engines for each Broker.

If you change the Engine Balancer on the Director, you must restart it. Also, all balancer settings must be equal on Primary and Secondary Directors. You can configure routing settings for online or offline Brokers.

Balancing and Service Discriminators

When the Director polls Brokers for Engine information, it also collects information about Service Discriminators and blacklisted Engines. The Director avoids a situation where Engines are at their home Broker and cannot take Services due to Service discrimination or blacklisting, but also it is not shared with another Broker. (Task-level discriminators are not taken into account.)

To report a Service discriminator, all of the following conditions must be true for a Service:

- The Service is not complete.
- The Service has pending tasks.
- The number of busy Engines working on the Service does not exceed the max Engines option of the Service.

To limit CPU and network usage during balancing, the maximum number of discriminators reported by each Broker can be configured on the Director. You can configure it at **Admin > System Admin > Manager Configuration > Engines and Clients > Max Service Discriminators**. This setting specifies the number of Services with discriminators per Broker that are considered.

If a Broker has more outstanding services with Service discriminators than the maximum specified, a message is logged in the Broker logs, similar to the following message:

```
INFO: [EngineSharing] Maximum number of discriminators 10 is reached when
collecting service level discriminators
```

When this occurs, the remaining discriminators over max (ten in the above example) are not considered when allocating Engines to this Broker. This does not prevent the Engines from being

reallocated to the Broker even if the Engines do not satisfy the discriminators. If Max Service Discriminators is set to 0, no discriminators are considered.

If the Max Service Discriminators is set to a very high value, reporting of balance data from the Broker to the Director takes more time. This slows down the balancing. If balancing is slowed down to an unacceptable level, lower the value of the parameter.

Engine Weight-Based Balancer

The Engine weight-based balancer allocates Engines to Brokers based on each Broker's Engine weight value. This value is the amount of Engines that are allocated to the Broker relative to the other Brokers' weights, when all Brokers are idle. The algorithm also considers session load and reallocates idle Engines to busy Brokers as they are needed. You can see a Broker's Engine weights value on the **Grid Components > Brokers > Broker Admin**page.

When it is necessary to restrict some Engines to a set of Brokers, the Engine weight-based balancer permits rule-based routing through Engine Properties. You can route Engines by their intrinsic properties, such as `cpuTotal`, and by user-defined properties. Create and assign user-defined properties with the **Grid Components > Engines > Engine Properties**page. Use the **Grid Components > Brokers > Broker Routing**page to set up routing rules based on these properties.

Home/Shared Balancer

The Home/Shared Engine balancer uses an algorithm in which an Engine has a set of Home Brokers that it always works on while it has outstanding tasks, yet the Engine can be shared with other Brokers when there are no outstanding tasks on any home.

The balancer uses Broker needs and Engine preferences for Brokers to allocate Engines to Brokers. Each Engine divides the existing Brokers into tiers (unordered sets of Brokers). The two default tiers are:

- The Engine's home Brokers
- The shared Brokers of those home Brokers

  You can introduce a third tier by splitting shared Brokers into two groups: preferred shared brokers and common shared Brokers. The higher the tier, the more the Engine prefers Brokers in that tier.

  The balancer uses the following rules:
- An Engine is routed to the highest-tiered Broker that has pending tasks. If multiple Brokers in the same tier have pending tasks, the choice is made at random, as if all weights were 1. The

number of Engines moving to a Broker is capped at the number of pending tasks if there are more pending tasks than available Engines in the tier.

- An Engine leaves its current Broker only if there is a needy Broker in a higher tier. An Engine does not move to a lower-tiered Broker unless it is idle.
- Failover Brokers are never allocated Engines unless they have pending tasks.
- If `Options.MAX_ENGINES` is set on a Service and the number of Engines from other Brokers are not shared with this Broker to run the Service. If there are more Engines on a Broker than the sum of all `MAX_ENGINES` values across active Services, the excess Engines are reported as available for sharing.

Use the GridServer Administration Tool to configure Brokers. Configure an Engine's home Broker with the **Grid Components > Engines > Engine Configurations**page. Configure Broker tiers (which Brokers share Engines with other Brokers) with the **Grid Components > Brokers > Broker Configuration**page as follows:

1. To set the first tier of Brokers, fill in the **Preferred Broker Sharing**field. Supply a comma-delimited list of the Brokers with which the current Broker shares Engines.
2. To set the second tier of Brokers, fill in the **Common Broker Sharing**field. Supply a comma-delimited list of other Brokers with which the current Broker shares Engines.
3. You can also define additional tiers of Brokers by delimiting them with semicolons in the **Common Broker Sharing**field. For example, to enter a second and third tier of Brokers, you can enter the list B1, B2, B3; C1, C2, C3.

For example, an Engine configuration's home Brokers are A and B. A's preferred broker is C; its common list is D,E. B's preferred Broker is F; its common list is G. An Engine with this configuration uses the following preferences: first: A, B; second: C, F; third: D, E, G. Within each tier, Brokers are equal, and ordering doesn't matter.

You can also use the Admin API to get or set the tiers. In `com.datasynapse.gridserver.admin.BrokerAdmin`, use the methods `setSharedBrokers` and `getSharedBrokers` to set or get the tier string. You can also create a Batch Definition that uses the Admin API to change the tiers according to a time schedule. For more information about using the Admin API, see the *GridServer Developer's Guide*.

On the **Grid Components > Brokers > Broker Configuration**page, you can also set a minimum number of idle home Engines for a Broker by adding the **Min Idle Home Engines**property column to the page. If the idle home Engine count is below this value, home Engines (idle or busy) are not logged off or shared to other Brokers.

Engine Balancer Configuration

To configure Engine Balancing on the Director, go to **Admin > System Admin > Manager Configuration > Engines and Clients**, and change the following properties:

SETTING	DESCRIPTION
Engine Balancer	The Engine balancer to use: Weight-Based or Home/Shared.
Rebalance Interval	The amount of time, in seconds, between balancing episodes. (Previously called the Poll Period.)
Logoff Timeout	The amount of time in seconds that an Engine waits to finish a task before logging off.
Broker Query Timeout	The maximum time to wait for Broker's reply for balancer data queries. The value is in milliseconds.
Soft Logoff	If true, Engine logoffs restart the JVM only when needed by a home Broker. This enables them to retain state and log in faster.
Engine Fraction	The fraction of extra Engines that moves to another Broker on a balance. This can be set to less than 1 to dampen Engine movement. For example, if the fraction is 0.5 and the balancer determines that a Broker has eight extra Engines, it moves four on the first balance. Assuming those Engines move, on the next balance it determines that there are four extra and moves two, and so on.
Engine Balance Maximum	The maximum number of Engines that can move to another Broker on a rebalance. The maximum applies over the entire grid. For example, if this property is set to 100 and the balancer determines to rebalance 200 Engines (after taking Engine Balance Fraction into account), then only 100 Engines are actually rebalanced.

SETTING	DESCRIPTION
Max Service Discriminators	The maximum number of service discriminators to consider when reporting balance data from each Broker.
Allow Routing When Sharing	If Broker Routing properties are used when the Sharing balancer is enabled. Under most circumstances, when using the Sharing balancer, it is best not to use routing.
Treat Pinned Engines as Busy	If pinned Engines are reported as busy from a sharing point of view. In some use cases, treating a pinned Engine as a busy Engine can reduce Engine fluctuation and improve overall grid performance. Note that this setting affects Engine sharing only. For information about Engine Pinning, see the *GridServer Developer's Guide*.

These settings must be identical on all Directors.

Engine Upper and Lower Bounds

You can configure upper and lower bounds on the number of Engines that can be logged in at a given time. Set these upper and lower bounds on the **Grid Components > Brokers > Broker Admin**page. If the columns for bounds are hidden, add them using the **Column**control. The minimum value specifies that the balancer algorithm always leaves at least this amount of Engines (assuming there are this many) on the Broker unless the Engines are needed by Brokers in higher sharing tiers. The maximum value is the cap on the total number of Engines to allow on the Broker. The balancing algorithms use both values.

Failover Brokers

A Failover Broker temporarily takes over running Service Sessions when the Client has no other Brokers to which it is permitted to connect. From the Client perspective, Failover Brokers become part of the pool of active Brokers when there are no other non-Failover Brokers on which the client is permitted. From the Engine perspective, a Failover Broker becomes part of the active pool when there are active sessions in progress on that Failover Broker. In either case, the algorithm now views this Broker as a non-Failover. It is important to take Failover Brokers into account when setting up the routing configuration. For example, if you are setting up a role to allow a client on

only one Broker under normal conditions, you must also include a Failover Broker in its Manager List if you wish this client to have a failover if its main Broker goes down.

See Grid Fault-Tolerance and Failover for more information.

Example Use Cases

This section describes example use cases of client routing and load balancing.

N+1 Failover with Weighting

An organization has four groups using all available Engines in a grid. One group has a guaranteed allocation of at least half of the grid anytime that it needs it, and the other three groups share the remaining Engines.

- Brokers - set up five Brokers. Each group gets a Broker, plus one is used for failover.
- Drivers - create four roles, one for each group. In each role, set the **Manager List**value to the group's Broker and the failover Broker. Assign the roles to the appropriate users.
- Engines - use the weight-based Engine Balancer. Adjust **Engine Weight**on the Broker Admin page so the first group's Broker is weighted at 3.0, and the other three groups' Brokers are weighted as 1.0. Set the failover Broker weight at 1.0, so that a group is not assigned any more resources than normal if their Broker goes down.

Engine Localization with Sharing

A company has two groups, one in New York and one in London. Each has a single middleware application that has a Driver that connects to its own Broker. Each group also has a set of CPUs that it expects to always be working on their own calculations. However, there are times when one group's Broker is idle, so they are allowed to share with each other.

- Brokers - set up four Brokers, a regular and a failover for each group. Each regular Broker shares with the other regular Broker, plus its own failover Broker.
- Drivers - create two roles, one for each group. In each profile, set the **Manager List**value to the group's Broker and its failover Broker. Assign the roles to the middleware application user.
- Engines - use the Home/Shared Engine Balancer. Set up two Engine Configurations, "London" and "New York", that set the Engines' homes to their respective Brokers.

In this scenario, the application always connects to its local Broker, unless it is down, in which case it moves to its failover. Whenever that Broker has pending requests, all of its Engines are always local. If the other group's Broker is idle, or if it does not need all of its Engines, any of its idle Engines are routed to the Broker that needs it.

Engine Configuration

Engine and Engine Daemon behaviors are controlled by a centralized profile called an *Engine Configuration*. When new Engine Daemons are installed, they use the default Engine Configuration for their platform (Windows, Linux, or other supported platforms.) Individual Engine instances take the configuration of their Engine Daemon. You can edit existing Engine Configurations, or create Engine Configurations for different subsets of Engines on your grid.

Editing an Engine Configuration

To change an Engine's settings, for example, to point it to a new Director, you can edit the Engine Configuration that is used by the Engine. Settings change on all Engines using that Configuration.

> **Note:**
>
> Changing an Engine Configuration causes Engines using that Configuration to restart.

> To change values in an Engine Configuration:

1. In the GridServer Administration Tool, go to **Grid Components > Engines > Engine Configurations**.
2. Select a configuration in the Configuration list. The list contains the names of all default Engine Configurations, plus any custom configurations that have been created.
3. Change any of the values that appear in the Engine Configuration.
4. Click **Save**to keep your changes, or **Cancel**to revert changes.

> **Note:**
>
> When adding values to the Environment Variables box, it is possible to set variables, particularly `PATH`, that can cause an Engine to fail to start.

Creating a New Engine Configuration

In some situations, you might need a subset of Engines to behave differently than other Engines on your grid. For example, you might want a large group of Engines to report to a different Director, or use a different compiler runtime. Instead of individually configuring each Engine, you can create

a new Engine Configuration with different Director settings, and assign the new Configuration to a subset of Engines.

To create an Engine Configuration:

1. Go to **Grid Components > Engines > Engine Configurations**, and click **Add**.
2. Select a platform for the Engine Configuration from the list.
3. Next to the selected platform, enter a name for the Configuration.
4. Click **Create**.

The initial values of the new Engine Configuration are the same as the default Configuration for the selected platform. You can then change the values of the configuration and click **Save**.

Copying an Engine Configuration

In a situation where you need several similar Engine Configurations, it's often faster to make a copy of an existing Engine Configuration, rather than creating many individual Configurations.

To copy an existing Engine Configuration:

1. Go to **Grid Components > Engines > Engine Configurations**.
2. Select an existing Engine Configuration from the list.
3. Click **Copy**. You are prompted for a name for the new Configuration.

Setting the Engine Configuration Used by Engines

To change the Engine Configuration of an Engine Daemon

Procedure

1. Go to **Grid Components > Engines > Daemon Admin**.
2. In the **Configuration**column, select an Engine Configuration from the list next to an Engine Daemon.

To change a large number of Engine Daemons' Configuration at once

1. Go to **Grid Components > Engines > Daemon Admin**.
2. Use the **Results Per Page**and **Search**controls to limit the table to only the Engine Daemons you want to change. For example, enter win32 in the search box and select **OS**as the column to search, then click **Go**. If you have more than twenty win32 Engines, enter 100 in the **Results**

Per Pagebox before you do your search. The goal is to get all of the Engine Daemons you want to change on one page.

3. In the **Actions**list, select **Configure Daemons on Page**.
4. The **Engine Daemon Editor**page opens, displaying all of the Engine Daemons previously in the table.
5. Select an Engine Configuration from the **Configuration**list.
6. Click **Save**.

Setting the Director Used by Engines

The Primary and Secondary Directors for an Engine are set during Engine installation. You can later change the Directors to which an Engine reports, by changing the Engine Configuration used by the Engine.

For Linux64 Engine

Prior to moving Engines from all versions earlier than 7.0.0_hotfix08 to 7.1.0, you must perform the following steps:

1. Stop the server.
2. As a best practice, take a backup of the existing `engine.sh` file.
3. Replace `engine.sh` file - from the `engineScriptUpdate` folder to `datasynapse/manager/webapps/livecluster/engineUpdate/linux64/ and datasynapse/manager-data/engineUpdate/linux64/` folder.
4. Start the server.
5. Let the Engines log in and then restart the Engine Daemon.
6. Move the Engines.

> **Note:**
>
> In case of Grid Architecture, perform the steps from step 1 to step 5 on the Primary Director.

Configuring an Engine's Director

1. Go to **Grid Components > Engines > Engine Configurations**.
2. Select the Engine Configuration used by the Engine. This is typically the operating system of the Engine.

3. In the **Directors and Brokers**section, change **Primary Director URL**and **Secondary Director URL**to the corresponding addresses and ports of the Primary and Secondary Directors, in the format `http(s)://address:port`.

Note that this changes the Directors for all Engines using that Engine Configuration. Also note that when moving Engines from one grid to another, Engines lose any custom Engine properties that are defined on the former Director's grid, because those are stored in the grid's administrative repository.

Configuring Engines With Multiple Network Adapters

In some network configurations, the host running an Engine can have more than one physical or logical network interface. When the Engine starts, it checks what IP address is in use and uses it to advertise its file server location. In cases where there is more than one network interface, it's possible for the Engine to pick the wrong one. To remedy this it is possible to force the Engine to choose a specific interface. To configure the Engine to use a different network interface, select the Engine Configuration it uses on the Engine Configuration page, and set the **Net Mask**value under the **File Server**heading to match the network range on which to run the Engine.

Configuring Engine Daemons to Use SNAT

You can configure Engine Daemons for use on the other side of a NAT from the Manager and Drivers. To do this, set the environment variable `DS_USE_SNAT_IP_ADDRESS` on an Engine Daemon to the IP address that you want it to report to the Manager and all Drivers as its address. Additionally, you must set **Self Ping**to `FALSE` in the Engine Configuration.

Using the System Classloader on an Engine

In most situations, the default classloader is used to load client classes. In some instances however, some applications might require the use of the system classloader. This requires deploying extra files to enable the Engine to use the system classloader.

To use the system classloader:

1. Copy the `DS_MANAGER/webapps/livecluster/WEB-INF/etc/DSEngine.jar` file to `DS_DATA/engineUpdate/shared`.
2. Go to **Grid Components > Engine > Engine Configuration****s**, select an Engine Configuration, and change the value of **Classloader**to **System**.

Note that this requires an Engine restart for any Grid Library that contains a `jar-path`.

Configuring a Global Shared Grid Library Directory

You can configure Engines to use a shared directory for Grid Libraries instead of Engines downloading their own copies.

To configure a global shared Grid Library directory:

1. Disable Engine synchronization on the **Admin > System Admin > Manager Configuration > Resource Deployment**page.
2. On the **Grid Components > Engines > Engine Configurations**page, set the **Grid Library Path**to a shared directory with read-only access that is available from all Engines.
3. Unpack all the necessary Grid Libraries into the shared Grid Library directory, with each library in its own subdirectory matching the archive filename.
4. Deploy the corresponding Grid Library archives to the Broker. The Broker still requires the same access to the Grid Libraries it always had, though Engines can now access them with an alternate method.

Notes:

- Grid Libraries do not automatically update when using this deployment method. Manually unpack and update the shared Grid Libraries, and restart the Engines.
- The shared Grid Library directory must contain both the Grid Library archive files and the extracted archives.
- Permissions are not maintained in `.zip` archives regardless of Broker OS.
- Permissions might be maintained when using `.tar`, `.gz` or `.tgz` Grid Libraries.
- If the configuration does not maintain permissions in any container or distribution Grid Libraries, you must ensure that the executable permissions are set properly for any scripts that run.

> **Note:**
>
> Using a global shared Grid Library directory with multiple Engines using the same network share can cause a significant drop in performance.

Configuring When Engines Run

You can configure GridServer to avoid conflicts between work and regular use of the machine. This is called adaptive scheduling, which you configure to adapt the Engine’s activity level to your

computing environment. The following section describes the methods used to configure when an Engine Daemon runs Engines. Unless otherwise indicated, all of the configuration options are available in Engine Configurations, which are modified on the **Grid Components > Engines > Engine Configurations**page.

Manual Mode

When Manual Mode is enabled in an Engine Configuration, the Engine Daemon runs at all times. This is the default. You can change an individual Engine Daemon to auto mode, or clear the **Enabled**option under **Manual Mode**on the Engine Configuration page to switch all Engines using that configuration to Auto Mode.

Auto Mode

Auto Mode lets you specify if Engines run on a computer, based on its utilization. For example, you might not want to run Engines when someone is logged in to the computer, or if its CPU utilization is above a certain percentage.

For Windows Engines, there are two methods for determining utilization: **User Idle**and **Processor Utilization**. When you select Auto Mode, select one of the two options. The default is User Idle. Auto Mode for UNIX Engines always uses Processor Utilization mode. In addition, you can also limit the hours an Engine runs by using the **Restricted Hours**settings. Each setting is described below.

User Idle

The User Idle feature is available for Windows Engines only. User Idle starts Engines if mouse and keyboard input have been idle for a given time on the computer hosting the Engine Daemon. This time is entered in seconds, and the default is 600 (10 minutes). Engines halt when there is mouse or keyboard input.

User Idle also has a percentage setting, which is used to determine when Engines stop running, in addition to UI activity. If processor utilization (in percent) on the host computer exceeds this value, Engines stop. To disable this feature, set it to 100%, the default.

Processor Utilization (Windows Engines)

The Processor Utilization option enables Engine Daemons to monitor system CPU usage, and start or stop Engines based on this statistic.

To use this option, enter two values: a percentage and a number of seconds. When CPU utilization drops below the given percentage for the given number of seconds, Engines start. For example, by default, Engines start when CPU utilization is below 50% for more than 30 seconds. When

utilization goes above the same CPU threshold value for the specified number of seconds, Engines stop.

CPU usage on single CPU systems is calculated as the total CPU usage, minus the CPU overhead of the Engine. Note that this does not include CPU usage for the Engine, invoke, or CPU idle processes. In multi-CPU systems where an Engine Daemon launches multiple Engine instances, CPU utilization is calculated independently for each CPU.

In multi-CPU systems, all Engine instances start and stop incrementally. This is called **Incremental Scheduling**, and is the default. When a CPU threshold value is reached, Engines start or stop one at a time. After an Engine starts or stops, there is a delay for a configurable interval—by default, the interval is 10 seconds. Utilization is checked again on the next CPU after the interval delay, and the process repeats.

If you do not select Incremental Scheduling, Engine Daemons use **Non-Incremental Scheduling**. In this case, all Engines on an Engine Daemon start or stop at the same time.

Processor Utilization (UNIX Engines)

On UNIX Engines, the Processor Utilization option works similar to its Windows counterpart, but there are some specific changes:

- You can configure both a starting and a stopping threshold and time. This lets you, for example, start Engines at 40% CPU utilization but stop them at 50%. (The Engines must also be above or below the CPU utilization percentage for the specified period of time.)
- The CPU usage sampling is averaged over a configurable period of time, with the default at 10 seconds.

When using incremental scheduling, Engines are started only when all other Engines are busy. This results in fewer Engine restarts when starting a new Service that requires a restart to load Grid Libraries, and restarts due to download of new libraries when idle.

Also, in non-incremental scheduling, CPU utilization is the average CPU utilization across all CPUs, and not individual CPU utilization. This total CPU utilization percentage is calculated by adding the CPU utilization for each CPU and dividing by the number of CPUs.

For example, if a four-CPU computer has one CPU running at 50% utilization and the other three CPUs are idle, the total utilization for the computer is 12.5%. Likewise, if the maximum CPU threshold is set at 25% on a four-CPU machine and four Engines are running, and a non-Engine program pushes the utilization of one CPU to 100%, all four Engines exit. Even if the other three

CPUs are idle, all Engines still exit. In this example, if the minimum CPU threshold is set at 5%, all four Engines restart when total utilization is below 5%.

This only applies to UNIX Engines, when non-incremental scheduling is used. With incremental scheduling on UNIX Engines, and in all scheduling on Windows Engines, each Engine Daemon only looks at the CPU utilization for the CPU on which it is running.

Similar to Windows, CPU utilization calculation does not include CPU usage for the Engine, invoke, or CPU idle processes.

Excluding Processes From Utilization Calculations

Windows and Linux Engines can also be configured to exclude a list of processes from utilization calculations.

The **Exclude the following list of processes from the processor utilization calculation**property in Windows and Linux Engine configurations can be used to specify processes that are ignored in utilization calculations. The value of this parameter must be a comma or semicolon-delimited list of case insensitive names, and are typically the executable names without extension.

Any process spawned by a Service must be in this list to prevent the Service from shutting down the Engine.

Note that Engine, invoke, and CPU idle processes are already excluded from utilization calculations by default.

Restricted Hours

When using Auto Mode, you can also specify a range of hours when Engines run. For example, if you want Engines to only run from 9:00 AM to 5:00 PM daily, configure this using Restricted Hours. You can also have GridServer ignore restricted hours settings on weekends.

To configure Restricted Hours:

1. Go to **Grid Components > Engines > Engine Configurations**.
2. Either select an existing Engine Configuration from the list to modify, or create a new profile.
3. Under the **Restricted Hours**heading, select the first checkbox.
4. Enter a time range when the Engines are allowed to run. For example, if you want Engines to run from 9:00 AM to 5:00 PM, enter 9:00 and 17:00.
5. Select the second option to ignore Restricted Hours settings on Saturdays and Sundays.

Restricted Hours settings do not apply to Manual Mode.

Configuring How Many Engines Run

There are three settings in the Engine configuration that determine how many Engine instances an Engine Daemon runs.

The `SMP Enabled` property specifies if the Engine configuration starts an Engine instance per processor core on the machine. This is set to True by default. Specifically, when SMP is enabled, the Engine Daemon runs a number of Engine Instances equal to the value of the `Minimum Engine Instances` property or the number of processor cores on the machine, whichever is higher. By default, this is set to 1, meaning the number of Engine instances always are the number of cores.

If SMP is disabled, the Engine Daemon always runs the number of Engine instances set in the `Minimum Engine Instances` property in the Engine configuration.

You can also adjust the number of Engine instances used, to reserve a certain number of cores for non-Engine activity. The value of the `Core Relative Engine Instances` parameter is added to the value specified in `Minimum Engine Instances`. The relative value can be a positive or negative number.

For example, if you want to ensure that a single core is always available with no Engine running on it, set the value of `Core Relative Engine Instances` to -1. If a 4-core machine uses this Engine configuration, its number of Engine instances is the higher of its number of cores and the minimum instances (4) plus the relative instances value (-1), or 3. If a 2-core machine uses this Engine configuration, it starts one Engine instance, leaving the other core free.

You can also manually specify the number of instances that run for an Engine Daemon. If you go to the **Grid Components > Engines > Daemon Admin**page, there is a list in the **Instances**column that enables you to select Default or auto SMP. You can type a number into this control to override the number of instances that are run.

Running Engines in Multiplexed Mode

On machines with multi-core processors, GridServer, by default, runs an Engine instance in its own process per logical CPU. While this enables each process to work on different Services, it also means each process runs in its own JVM, and requires its own set of application data. This can cause unwanted overhead when running the same Service across many CPU cores.

To address this issue, you can run Engines in multiplexed mode, meaning that all Engine instances run in a single process. This enables all Engine instances on that machine to share the same set of application data, and multiplex their communication with the Broker.

Multiplexed Engines appear as individual Engines in the GridServer Administration Tool and by the Admin API. Each multiplexed Engine has its own log, work, temp, and data directories. They produce the same output to the reporting database.

You can’t mix single-process Engine instances and multiplexed Engines on the same machine. For example, on an eight-core machine, you can’t have two processes with four multiplexed Engines each; you can only have eight multiplexed Engines in one process.

Multiplexed Engines have the following differences with Engines running in their own processes:

- All Engines are logged in to the same Broker at any given time. This is enforced by the balancer.
- All Engines work on the same Service session at any given time. This is enforced by the Scheduler
- All Engines reside in the same classloader, which means they can all share the same set of Service Session data.
- They share the same initialization data and Service object. The Service init method is only called once, and all updates are synchronized across all Engines.
- If one Engine has to log off, all Engines in the process are also logged off with the same reason.
- Incremental scheduling is not supported.
- Autopack is not supported.
- The Service implementation must be threadsafe. See the *GridServer Developer’s Guide* for more information about multiplexed mode development considerations.
- Running more than one task from a recursive Service is not supported.
- All multiplexed Engines share one set of properties. For example, getting the instance property always reports instance 0, regardless of the Engine instance.

When using multiplexed Engines, Engine resources are shared. The first Engine does all library loading, and this Engine is the first to start work. This also means that Engine Hooks are only loaded once, as hooks are loaded when Grid Libraries are loaded.

When using GridCache, there is one cache per process, and all communication is handled by Engine instance 0.

Communication and Task Scheduling

When Engines are configured as multiplexed, communication with the Broker is multiplexed over a single HTTP keep-alive session to reduce overhead. Also, only one heartbeat is communicated per process. Heartbeats are received by Engine instance zero and simulated to the other Engine Proxies.

To prevent starvation of Engines to other Service sessions, a lease duration is defined. This is the amount of time a set of multiplexed Engines work on one Service before they stop working on that Service and are allowed to work on others if necessary.

Given a lease duration of 5 minutes, the lifecycle for this is as follows:

- When work is started on the Grid, Engine instance 0 picks up a task.
- Immediately following that, all other instances pick up tasks from the same Service session.
- Assuming there is enough work, the instances continue to take tasks from that session.
- After 5 minutes have passed, when any instance finishes a task, it does not pick up more tasks.
- As soon as the last instance has finished, immediately after the response has been written, Engine instance 0 requests another task from any appropriate session. It can be the same session or another session.

Configuration

To enable multiplexed mode, set the Multiplexed Mode property to true in the Engine configuration. The default value is false.

To set the lease time, go to **Admin > System Admin > Manager Configuration > Services**, and set the value of the `Multiplexed Engine Lease` property. The default time is one minute.

The `MULTIPLEXED_MASTER_ID` Engine property is set on any multiplexed Engine to contain the `sessionID` of Engine instance 0 within its process.

There are two buffer timeouts:

- `MULTIPLEXED_ENGINE_MAX_BUFFER_WAIT`: This is an absolute time set per Service.
- `MULTIPLEXED_ENGINE_AVG_BUFFER_WAIT`: This is a factor that is multiplied by the average task duration or the tasks that run during the lease. The default is 0.5. So if the average duration is 10 seconds, this timeout is 5 seconds.

The timeout is measured starting from the first send request.

Configuring 64-bit Engine Daemons to run 32-bit Services

The 64-bit Windows Engine Daemon can be configured to allow running 32-bit Services. If allowed, when an Engine takes such a task, it clears all existing loaded libraries and restarts as a 32-bit process.

Configuration

On the Windows Engine Configuration page, there is a setting called **Additional Platforms**under **Classes, Libraries,**and **Paths**. It is a list with two values, `None` and `win32`. By default it is `None`. Setting it to `win32` on a Windows 64-bit configuration enables the behavior. Setting it to a Windows 32-bit configuration has no effect. This sets an Engine property named `additionalPlatforms` to `win32` or an empty string.

When this property is enabled, Engines download any Grid Libraries with the OS set to **win32**in the root element, in addition to all others it normally syncs.

Note that because Engines take tasks only if they have the root Grid Library of a Service downloaded, this inherently allows these Engines to take 32-bit Windows Services.

When this property is enabled, the following behavior occurs:

- When a 32-bit task is taken by a 64-bit Engine, it restarts as a 32-bit Engine, clearing all Grid Libraries except for the one from the Service and its dependencies
- When a 32-bit task is taken by a 32-bit Engine, it loads normally.
- When a task that is anything but 32-bit is taken by a 32-bit Engine, it restarts as a 64-bit Engine, clearing all Grid Libraries except for the one from the Service and its dependencies.
- When a task that is anything but 32-bit is taken by a 64-bit Engine, it loads normally.

Specifying that a Service is win32

Any Service for which the root Grid Library is marked as `os="win32"` is considered to be a win32 Service.

If a Service's root Grid Library is not marked as such, the Engine remains 64-bit and fails.

Routing 32-bit Tasks to 64-bit Engines

By virtue of the Engine having the 32-bit Grid Library, they by default are allowed to run such Services by the scheduler.

Note that existing applications might not use the OS attribute on the root Grid Library; rather they might use discrimination instead to route Services to win32 Engines. To use 64-bit Engine Daemons on these Services, you must set the OS attribute on that Grid Library.

Configuring a Caching HTTP Proxy Server

Consider a scenario where a GridServer deployment where a Broker and its Engines are separated by a WAN. In this case, it can be inefficient to transfer the same data over the WAN to multiple Engines from the Broker or the Clients. One solution is to use an HTTP proxy server (such as Squid Web Cache) to cache the session's init data. Any Engine that works on the session must transfer this init data. You can specify a proxy server in an Engine configuration, and the proxy server caches the Service data for other Engines also using the same proxy server.

To use a proxy server, such as Squid, for resource synchronization or data transfer, configure `Proxy Host` and `Proxy Port` parameters to the proxy hostname and port. In case you want to use Proxy with authentication along with `Proxy Host` and `Proxy Port`, you must use `Proxy Username` and `Proxy Password`. HTTPS proxy server uses the `Proxy Scheme` field.

The HTTP Proxy Scheme uses the `FORWARD` Proxy concept whereas the HTTPS uses the `REVERSE` Proxy Concept.

The HTTP Proxy Scheme supports `Basic` and `Digest` authentication and HTTPS supports `Basic` authentication.

For NON-SSL servers, Proxy uses the HTTP scheme whereas for SSL server Proxy uses the HTTPS scheme.

The `Use Proxy for Data Transfer` causes Engines to use the HTTP or HTTPS proxy server for the download of any session's init data. The `Use Proxy for Resource Synchronization` property causes Engines to use HTTP or HTTPS proxy for resource synchronization download.

If Engines are configured to use a proxy server and the proxy is not available, the Engine does not attempt to download the Service data using alternate connection parameters. It is the administrator's responsibility to ensure that the proxy server is up and properly configured. The administrator must consider implementing DNS or IP failover if high availability is required.

Token security for all resource downloads is not supported when using a proxy server. It is assumed that the proxy server is in the same LAN and the LAN is secure. Note that the proxy server cannot download a resource until one of the Engines provides a valid download token.

Note that the following potential problems might occur when using caching proxy servers for resource synchronization:

- Engines can download stale copies of an updated resource from the proxy server. This occurs when the cache timeout is too long or a recently downloaded resource is updated shortly afterward.
- The proxy server might not be able to serve a cached copy of an updated resource if multiple Engines start downloading the resource in a very short time span. This can occur when using Squid even if the `collapsed_forwarding` parameter is enabled.

  To address these issues, adjust the cache size to ensure that the proxy server can cache large files. Additionally, analyzing the resource download pattern and the proxy server configuration is recommended to achieve optimal results. The important factors to consider are:
- Size of the total resources that require downloading
- Size of the total proxy cache, the maximum size per object
- Maximum age of a cached object
- Proxy server behavior for concurrent requests
- Configure the HTTP proxy server to ignore the no-cache header. For example, the following `refresh_pattern` option in the `squid.conf` file causes the squid cache to ignore no-cache headers for URLs matching the regular expression of "`^http://.*/livecluster/resourcesproxy`":

  ```
  refresh_pattern ^http://.*/livecluster/resourcesproxy 0 20% 4320 ignore-no-cache
  ```

Configuring an External Engine Daemon Admin Tool

It is possible to configure an external administration or management tool or system that can be opened from GridServer's Administration Tool. When configured, you can open the tool from the Engine Daemon Admin page with an item in the Actions control. Selecting the new item opens a URL to the external tool. A set of runtime macros enable you to customize the URL further.

To configure an external Engine Daemon Admin Tool:

1. In the GridServer Administration Tool, go to **Admin > System Admin > Manager Configuration > Engines and Clients**.
2. In the **Daemon Admin External Tool URL**box, enter the URL to your external tool.

3. The following runtime macros can be used: `${DaemonID}`, `${UserName}`, `${IP}`, `${Status}`, `${PrimaryDirector}`, `${SecondaryDirector}`, and `${OS}`. For example, http://example.com/grid/reports.py?${DaemonID} might be used to look up reports for an Engine Daemon with an external query tool.
4. In the **Daemon Admin External Tool Name**box, enter a description of the configured tool.

To use an external tool, select it from the Actions control next to an Engine Daemon on the Engine Daemon Admin page. It is named "Link to" plus the name you defined in the **Daemon Admin External Tool Name**box.

Quarantine Brokers

In a security-oriented environment, it can be necessary to prevent new or untrusted Engines from joining a grid and downloading potentially sensitive application data or resources until the GridServer administrator explicitly grants permission for them to join. You can exercise this control by using a *Quarantine Broker*, a dedicated Broker in a grid, used only for Engine staging and verification. When Engines do not have permission to log in to other production Brokers on your grid, they can only log on to the Quarantine Broker and await permissioning by an administrator.

The quarantine status is set on any Engine Daemon with the Administration Tool or Admin API. If an Engine Daemon’s quarantine status is set to “Verified”, the Engines managed by the Daemon might log in to the production Brokers after an Engine Daemon restart. The Engines managed by a quarantined Engine Daemon might only log in to the Quarantine Broker.

Quarantine Broker Concepts

To use a Quarantine Broker, there are two components: the Quarantine Broker, and a method to *Set Quarantine* (send Engine Daemons to the Quarantine Broker) and *Clear Quarantine* (change Engine Daemon status to allow Engine Daemons to log onto production Brokers.)

The Quarantine Broker is specified on the Director. When the Quarantine Broker is specified, the Director allows individual Engine instances to log in to either the production Brokers or the Quarantine Broker based on the value of quarantine status of the managing Engine Daemon. If the quarantine status of an Engine Daemon is “Verified”, all Engines managed by the Engine Daemon might only log in to the production Brokers. Otherwise, the Engines might only log in to the Quarantine Broker.

When using Engine balancing and a Quarantine Broker, the balancing only occurs on the production Brokers. The Quarantine Broker is ignored for the purposes of Engine balancing, and only verified Engines are balanced between the production Brokers.

If the Quarantine Broker is not set in a grid, Engines might log in to any Brokers in the grid allowed by routing and balancing configurations.

Quarantine Status on Engines

The Director determines if an Engine is quarantined by looking for an Engine property called `QuarantineStatus`. The `QuarantineStatus` property value is set to “New Engine” on all newly installed Engines. This ensures that all new Engines are quarantined upon installation when there is a quarantine Broker on the Grid.

When an Engine Daemon has its `QuarantineStatus` property set to any other string than “Verified”, including the null string or having the property missing, it is considered quarantined, and its Engines are only allowed to log in to the Quarantine Broker. When an Engine Daemon has its `QuarantineStatus` property set to the string “Verified”, it is verified, and its Engines are now allowed to log in to other Brokers as per its routing rules.

An unverified Engine Daemon might be cleared from the Administration Tool by changing the `QuarantineStatus` property. This can be automated by using the Admin API. A verified Engine Daemon can be quarantined similarly. There is also an API method that can be called in a Service Session to quarantine an Engine Daemon.

After quarantine status change, the Engine Daemon needs to be restarted for the managed Engines to log in to the intended Broker set. This restriction minimizes the risk of setting the quarantine status by mistake.

Requirements

All Directors require the same Quarantine Broker setting. When failover is configured, this means the Secondary Director has the same Quarantine Broker configuration if the Primary Director fails.

Although a grid might have multiple Brokers, either for redundancy or volume, you can only define one Quarantine Broker in a grid.

When using a Quarantine Broker, you must have an account with **Manager Configure Edit**access in its Security Role (only available in the Configure role by default) to clear quarantine status of an Engine Daemon or set or change the Quarantine Broker definition. An account must have **Engine Properties Edit**access in its Security Role (available in Configure and Manage roles by default) to set Engine Daemon quarantine status. Any role with **Engine Daemon View**access (all roles by default) can see Quarantine Broker and Engine Daemon quarantine status.

Configuring a Quarantine Broker

To add a Quarantine Broker:

1. Install an additional Broker, if you don't already have an extra one installed.
2. Determine the name of the Broker you wish to use. Broker names are automatically given at installation, and are typically numeric. In the GridServer Administration Tool, **Grid Components > Brokers > Broker Admin**shows a list of your Brokers, with their names in the **Broker Name**column. Find the name of the Broker you wish to use it for a Quarantine Broker. Note that you can also use this page to change the names of Brokers, if you want to give the Broker a more logical name.
3. Go to **Admin > System Admin > Manager Configuration > Engines and Clients**. Under the heading **Quarantine Broker**, enter the name of the Broker you want to be the Quarantine Broker, then click **Save**. You do not need to restart the Manager.
4. Repeat this configuration (with the same Broker name) for both Primary and Secondary Directory on your grid.

The Quarantine Broker settings persist after Manager restart or future Manager upgrade. If you must change or remove a Quarantine Broker, repeat step 3, but enter a new or blank value for the name.

Setting Quarantine Status on Engines

There are two ways to set and clear quarantine status: interactively with the GridServer Administration Tool, or programmatically with the GridServer Admin API. You can also use the API to self-quarantine Engines. Each method is described below. Also, the following constraints apply when setting quarantine status:

- Due to XML limitations for Engine properties, special XML chars are not allowed in the `QuarantineStatus` property.
- If there is a problem setting the `QuarantineStatus` property, it throws an `AcccessException` (a subclass of `AdminException`) based on the new value.

Using the GridServer Administration Tool

Use the Administration Tool to set or modify the `QuarantineStatus` property in one of the following ways:

- From **Grid Components > E****ngines > Daemon Admin**, select **Edit/View Properties**from the **Actions**list. Select **QuarantineStatus**from the **Properties**column, and enter a new value.

- From **Grid Components >****Engines > Daemon Admin**, select **Set Property for Daemons on Page**, then select **QuarantineStatus**from the Engine Property List, and change the value of the Property `.`

Using the Admin API

The following method in the `EngineDaemonAdmin` Admin API interface can be used to set and clear Engine Daemon quarantine status:

```
public String setProperty(long id, String key, String value) throws
Exception
```

The value of `key` must be `QuarantineStatus` to set the Engine Daemon quarantine status. The value `id` is the identifier of the interested Engine Daemon. The value might be retrieved by calling `EngineDaemonAdmin.getEngineDaemonIds()`.

The corresponding API call for `setProperty` are also available for C++ (`dsdriver::EngineDaemonAdmin.setProperty`) and .NET (`EngineDaemonAdmin.SetProperty`)

Engine Self-Quarantine Using an API Call

You can also quarantine an Engine using the following method in the Java API:

```
com.datasynapse.gridserver.engine.EngineSession.quarantine (String reason)
throws GridServerException;
```

This method might be called in a Service Session. When this method is called, a synchronous message is sent to the Broker, the Broker then forwards the message plus the Engine session ID to the active Director synchronously. When the Director receives the message, it finds the Daemon ID based on the Engine session ID and sets the Engine property. The Director asynchronously restarts the Engine Daemon identified by the Daemon ID. If any exception occurs, it propagates back to the Engine. Due to the fact that the Engine ID is not passed from the Engine to the Broker, this API call can not quarantine any other Engine Daemons besides the managing Daemon of the current Engine.

The affected Engine Daemon restarts for the quarantine status change to take effect. When the affected Engines receive the restart command, any running Services are canceled, as usual.

The corresponding API call for `quarantine` are also available for C++ (`dsdriver::EngineSession.quarantine`) and .NET (`EngineSession.quarantine`)

Quarantine Broker Constraints

There is no finer configuration to direct Engine instances to specific Brokers using this feature.

Due to the fact that there is only one quarantine Broker defined in a grid, several issues might arise as a result:

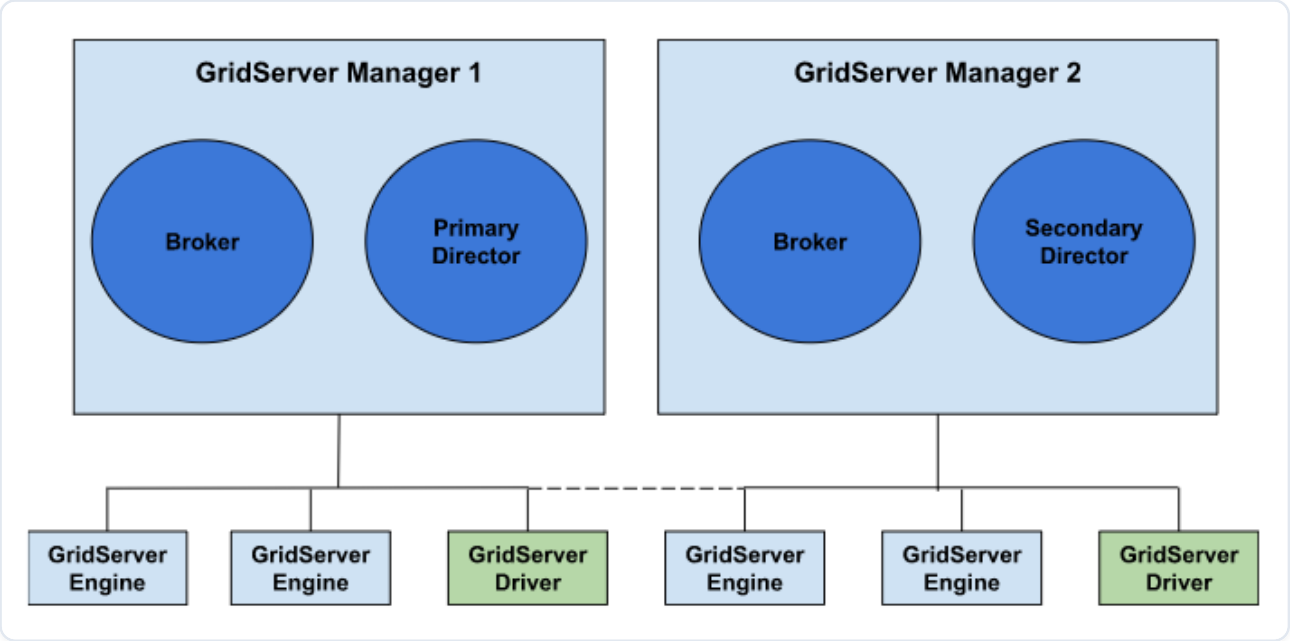
- Quarantined Engines that don't collocate with the quarantine Broker need to send traffic over the WAN.
- If the Quarantine Broker is not up or accessible, quarantined Engines cannot log in to any Brokers in the grid. This might generate heavy login retry load when the network is partitioned or unstable.
- Drivers are not treated in any special manner with regard to routing to Quarantine Brokers. This is because some might wish to run tests that require a Driver on Quarantined Engines. Routing rules are required to prevent production Drivers from logging in to Quarantine Brokers.

Grid Fault-Tolerance and Failover

GridServer is a fault-tolerant and resilient distributed computing platform. The GridServer platform recovers from a component failure, and guarantees the running of Services over a distributed computing grid with diverse, intermittent compute resources. This section describes what GridServer does in the event of Engine, Driver, and Manager failure. Failures of components within the grid can happen for a number of reasons, such as power outage, network failure, or interruptions by end users. For the purposes of this discussion, failure means any event that causes the grid components to be unable to communicate with each other.

The Fault-tolerant GridServer Deployment

A DataSynapse GridServer deployment consists of a Primary Director, an optional Secondary Director, and one or more Brokers. Drivers and Engines log in to the Director, which routes them to one of the Brokers. Directors balance the load among their Brokers by routing Drivers and Engines to currently running Brokers.



A minimal fault-tolerant GridServer deployment contains two Directors, a Primary and a Secondary, and at least two Brokers. The Brokers, Engines, and Drivers in the grid have the network locations of both the Primary and the Secondary Directors. During normal operation, the Engines and Drivers log in to their Primary Director; the Secondary Director is completely idle.

Other GridServer topographies, such as having multiple Managers to handle volume or to segregate different types of Services to different Managers, are discussed in the *GridServer Installation*.

Heartbeats and Failure Detection

Lightweight network communications sent at regular intervals, called *heartbeats*, are sent between GridServer components, such as from Drivers to Brokers, from Engine Instances to Brokers, and from Engine Daemons to Directors. A Manager detects Driver and Engine failure when it does not receive a heartbeat within the configurable heartbeat interval time. Drivers detect Broker failure by failing to connect when they submit tasks or poll for results. Engines detect Broker failure when they attempt to report for work or return results. To minimize unnecessary messaging, a heartbeat is sent only if no other message has been sent within the heartbeat interval.

Heartbeat period for clients can be configured in the GridServer Administration Tool at **Admin > System Admin > Manager Configuration > Communication**.

Manager Stability Features

Several precautions prevent Manager failure due to excessive traffic. For instance, the number of threads used for file update is limited. This prevents a large number of file updates from Brokers to Engines from preventing other HTTP activity due to use of all of the HTTP threads on the

application server. Instead, Engines retry the download later when this maximum is reached. By default, this is set at 50 threads, but can be changed in the GridServer Administration Tool; go to **Admin > System Admin > Manager Configuration > Communication**, and change the `Maximum Resource Download Connections` property.

The number of Broker/Director messaging threads is also limited. If this limit is reached, clients retry rather than immediately fail.

Engine Failure

Network connection loss, hardware failure, or errant application code can cause Engine failure. When an Engine goes offline, the work assigned to it is queued again and assigned to another Engine. Although work done on the failed Engine is lost, the task is assigned to a new Engine. Engines that have built up a considerable state or cache or that are running particularly long tasks can cause a larger loss if Engine failure occurs. This can be avoided by shortening task duration in your application or by using the Engine Checkpointing mechanism. For more information about task duration, see Optimizing the Grid.

Each Engine has a checkpoint directory where a task can save intermediate results. If an Engine fails and the Manager retains access to the Engine machine's file system, a new Engine copies the checkpoint directory from the failed Engine. It is the responsibility of the client application to handle the correct resumption of work given the contents of the checkpoint directory.

Note that if an Engine Daemon logs off the Director or otherwise fails, it does not log off its Engines. Provided the failure has not caused the Engines to also fail, they continue working and return results when completed.

Driver Timeout and Failure

When a client application fails, the Broker detects the failure when the Client does not send a heartbeat and does not log back in within a specified time.

This time is defined as the `Client Timeout Minutes` plus the Driver Heartbeat Timeout. The Driver Heartbeat Timeout is the `Max Millis Per Heartbeat` property times the `Timeout Factor` property. Note that for Max Millis Per Heartbeat, this setting is the maximum value. The actual value is randomly between half this value and the value. (For example, the default value is set at 15000 milliseconds. This means the value is between 7500 and 15000 milliseconds.)

For example, by default, `Client Timeout Minute` is 5 seconds (300,000 ms), `Max Millis Per Heartbeat` is 16,000 ms and **Timeout Factor**is 15. This means that the client timeout is between 300,000 + 8,000 * 15 and 300,000 + 16,000 * 15 ms.

The Client Timeout Minutes is set on the Manager at **Admin > System Admin > Engines and Clients > Client Management > Client Timeout Minutes**. The Driver Heartbeat Timeout the `Max Millis Per Heartbeat` property times the `Timeout Factor` property. Both are set at **Admin > System Admin > Communication > HTTP Connections > Driver Heartbeat**.

When client timeout happens, any currently running Services are canceled. If this happens, application failure recovery or restart is the responsibility of your application.

The exception to cancellation is fully submitted Services of type `Collection.LATER` or `Collection.NEVER`. Also, if a Client is collecting results from a `Collection.LATER` type of Service, none of the outputs are removed until all are collected and the Client destroys the Service, so that if a Client fails during collection it can restart and recollect the outputs.

It is also possible to collect tasks from a failed client application for Services of type `Collection.AFTER_SUBMIT` and `Collection.IMMEDIATELY` if cancellation has not yet occurred. To do this, you must collect and pass in the Driver session ID, in the same fashion as the collection of a `Collection.LATER` Service. For more information about the collection of `Collection.LATER` Services, see "Deferred Collection (Collect Later)" in the *GridServer Developer's Guide*.

All Driver file servers return a Server Unavailable code with instructions to retry if they are processing too many concurrent requests. This significantly reduces the chance of a Service invocation failing due to a temporarily overloaded Driver.

Director Failure

If the Primary Director fails, the Secondary Director takes over balancing and routing Drivers and Engines to Brokers. The Directors do not maintain any state, so no work is lost if a Director fails and is restarted. Also, both Directors follow the same rules for routing to Brokers, so it makes no difference which Director is used to log in.

The Primary Director is also responsible for an internal database, which contains data needed by the Manager for operation, such as the User list, routing properties, and so on. You can modify these values only on the Primary Director. This database synchronizes with the Secondary Director while both are running. The Secondary Director backs up this database on every database backup so that the grid can remain in operation when the Primary Director is down. Features that modify the internal database are not available from the Administration Tool when a Secondary Director is active.

Broker Failure

Like the Director, the Broker is a robust application that runs indefinitely. The Broker typically only fails when there is a hardware failure, power outage, or network failure. However, the fault tolerance built into the Drivers guarantees that all Services complete even in the event of failure.

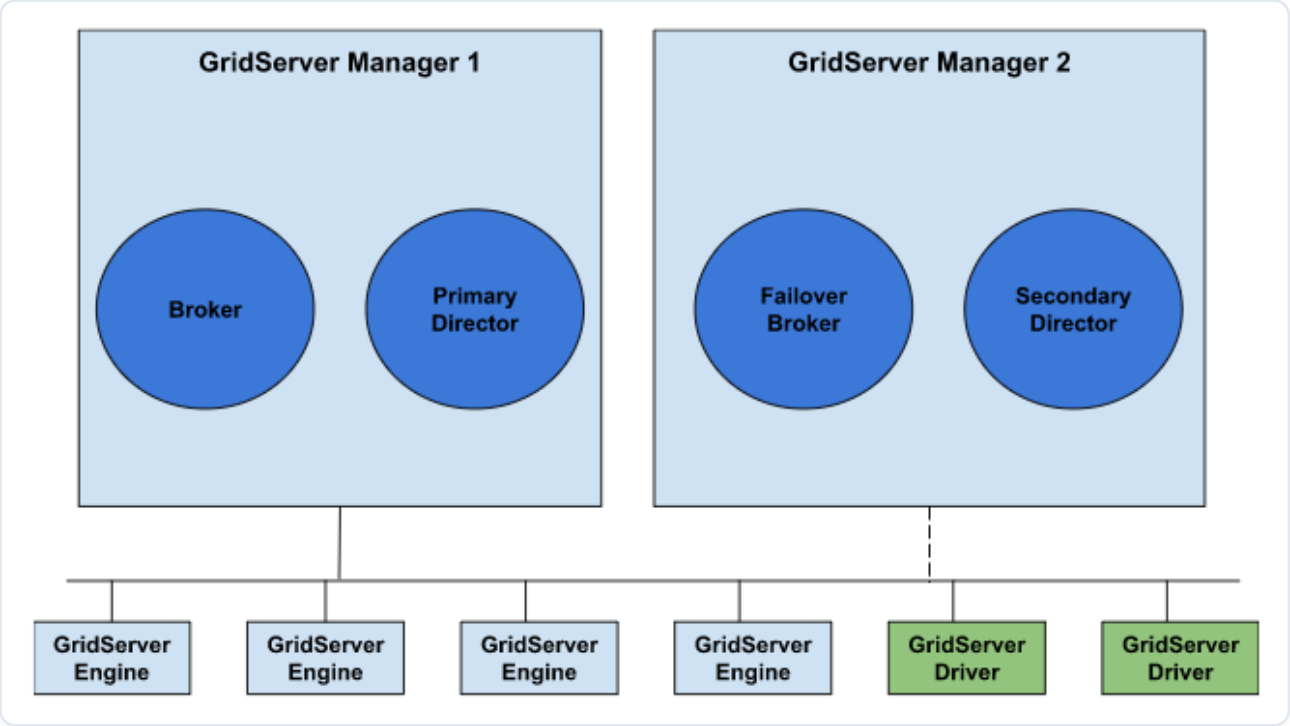
The most likely reason that a Driver disconnects from its Broker is a temporary network outage. Therefore, the Driver does not immediately attempt to log in to another Broker. Instead, the Driver waits a configurable amount of time to reconnect to the Broker to which it was connected. After the configured wait time expires, the Driver attempts to log in to any available Broker. Specify the configured amount of time as `DSBrokerTimeout` in the `driver.properties` file. The property is `BROKER_TIMEOUT` in the API.

After the Driver times out and reconnects to another Broker, all Service instances resubmit any outstanding tasks and continue. Tasks that are already complete are not resubmitted. The Service instances also resubmit all state updates in the order in which they were originally made. From the Service instance point of view, there is no indication of error, such as an exception or failure. There is just the absence of any activity during the time in which the Driver is disconnected. That is, all Services run successfully to completion as long as eventually a suitable Broker is brought online.

If an Engine is disconnected from its Broker and there are no Failover Brokers, the process shuts down, restarts, and logs in to any suitable Broker. Any work is discarded. Failover brokers are described in the next section.

Failover Brokers

In the fault-tolerant configuration, you can set up some Brokers as Failover Brokers. When a Driver logs in to a Director, the Director first attempts to route it to a non-Failover Broker. If no non-Failover Brokers are available, the Director considers all Brokers, and typically routes the Driver to a Failover Broker.



GridServer configuration with Failover capability

A Failover Broker without active Services is not considered for Engine routing. If it has active Services, it is considered like any other Broker, and follows Engine routing like any other Broker. Thus, if a Failover Broker becomes idle, its Engines are routed back to other Brokers.

The Primary Director monitors the state of all Brokers on the grid. If a Driver that is logged in to a Failover Broker is able to log in to a non-Failover Broker, it is logged off so it can return to the non-Failover Broker. All running Services continue on the new Broker by auto-resubmission.

By default, all Brokers are non-Failover Brokers. Designate one or more Brokers within the grid as Failover Brokers when you want those Brokers to remain idle during normal operation.

Task Fault Tolerance

Task fault tolerance enables an Engine to continue executing a task even if it logs off of a Broker, so that it does not lose work due to a Broker failure. This means that if an Engine is working on a task, and it logs off the Broker, it does not immediately exit. Rather, it continues to work on that task, while continuing attempts to log in to a Broker with the Service on which it is working. If it does not log back in within a defined time period, it exits. If it does log back in, it notifies the Broker that it is working on the task. If the task is completed, it immediately sends the result; otherwise, it does so upon completion.

Using this feature is only recommended when you have individual tasks that take many hours to finish. For example, if a report runs during the night and some tasks take eight hours to process, task fault tolerance ensures that the eight-hour tasks don’t have to start from the beginning if the

Broker fails at 7 AM. Enabling task fault tolerance can diminish the efficiency of the grid, since it redundantly schedules all outstanding tasks. For short tasks, it's usually more efficient to simply recalculate tasks in the event of a Broker failure.

Consider the following example of task fault tolerance:

1. An Engine and Driver are connected to Broker A. The Driver submitted a Service, and an Engine is working on that Service.
2. Broker A goes down.
3. The Driver tries to reconnect with Broker A. The Engine continues working and tries to reconnect to Broker A.
4. After five minutes, the Driver gives up attempts to connect to Broker A. It connects to Broker B and resubmits outstanding work.
5. The Service is now on Broker B. The Engine logs in to Broker B and indicates that it is taking that task. If the Engine already finished its work, it immediately writes the task. Otherwise, after it completes its work, it writes its task.

If another Engine takes the task by the time the original Engine logs in, no attempt is made to cancel the task on the Broker. It is the same as a redundantly rescheduled task.

The situation is similar when an Engine logs into a Failover Broker and works on a task. When the Driver switches back to the Primary Broker, the Engine logs off the Failover Broker and reconnects to the Primary Broker. The task is not canceled.

To enable task fault tolerance, go to **Admin > System Admin > Manager Configuration > Engines and Clients**, and change the value of **Engine Timeout Minutes**. Make the Engine timeout longer than the Driver's timeout, which is the value of `DSBrokerTimeout` set in the `driver.properties` file (five minutes by default.) Note that changes to this value take effect at the next Engine Login.

To use task fault tolerance, another Broker must be available for failover, and the Client running the session must fail over to the Broker and resubmit its session.

No attempt is made upon login of the Engine running a fault-tolerant task to cancel that same task if it has already been taken by another Engine.

Batch Fault-Tolerance

Batch Schedules that exist on a Manager are persistent, provided the **Next Run**field is not `never`. This persistence provides failover capability in the event of a Manager failure, because the Batch Schedules still exist when the Manager is restarted.

The following Batch Schedules are persistent:

- Absolute schedules
- Relative schedules with repeat
- Cron schedules

All persistent Batches restart when the Manager restarts, as if they are scheduled for the first time. Batch runs scheduled for the time when the Manager is down are ignored.

GridCache and PDriver Fault-Tolerance

GridCache supports fault tolerance as described below. Note that Primary and failover Brokers must have their clocks synchronized for GridCache failover.

Because PDriver uses GridCache for its implementation, the same fault-tolerance capability described below applies to PDriver jobs.

Client

If any client puts data in the cache and subsequently fails or logs out, that data is still available to all other clients. This is because the Broker maintains the master index and a complete view of the cached data. This availability does not apply to the local caching mode where a region has a local loader that does not synchronize with the other local caches.

Broker Restart

You can configure GridCache to survive Manager restart and failure. GridCache’s cache index is rebuilt on system startup; objects persisted on the Broker’s file system are recovered. If some or all of the cache is stored in memory, that information is lost.

Failover

A failover Broker can manage a GridServer cache when a regular Broker goes down, if the persistent cache directory is on a shared filesystem. Configure the location of this filesystem from the Manager Configuration page in the GridServer Administration Tool. When the regular Broker goes down and the failover Broker takes over, the failover Broker builds its cache index and begins

managing the cache from the shared filesystem. All clients that then fail over to the failover Broker can get references to the existing cache regions on the shared file system.

Note that a failover Broker can be configured to fail over only to one shared cache directory. Therefore, a failover Broker cannot serve as a failover for multiple Brokers with different cache directories; a different failover Broker has to be used for each Broker.

Note that a failover Broker can fail over to *only one* shared cache directory. One failover Broker cannot serve as a failover for multiple Brokers that have different cache directories. Brokers with different cache directories require separate failover Brokers.

Administration and Maintenance

This section contains several procedures that are commonly used when administering a GridServer Manager. Most of the tasks outlined below use the Administration Tool. The Administration Tool also provides online help, which further describes each page's features.

Configuration Issues

Installation on Machines With Multiple Network Adapters

In some network configurations, on a machine with more than one network interface, a GridServer component can default to using the incorrect interface. You can configure the component to use the correct interface as follows:

- Drivers: To configure a Driver to use a different network interface, edit the `driver.properties` file and set the `DSLocalIPAddress` property to the IP number of the correct interface. For example:

  ```
  DSLocalIPAddress=192.168.12.1
  ```

- Engines: To configure an Engine to use a different network interface, select the Engine Configuration that is used by the Engine on the Engine Configuration page, and set the **Net Mask**value under the **File Server**heading to match the network range on which to run the Engine.

Configuration Issues

Installation on Machines With Multiple Network Adapters

In some network configurations, on a machine with more than one network interface, a GridServer component can default to using the incorrect interface. You can configure the component to use the correct interface as follows:

- Drivers: To configure a Driver to use a different network interface, edit the `driver.properties` file and set the `DSLocalIPAddress` property to the IP number of the correct interface. For example:

  `DSLocalIPAddress=192.168.12.1`

- Engines: To configure an Engine to use a different network interface, select the Engine Configuration that is used by the Engine on the Engine Configuration page, and set the **Net Mask**value under the **File Server**heading to match the network range on which to run the Engine.

Using UNC Paths in a driver.properties File

To use UNC paths to specify a hostname or directory within a `driver.properties` file, you must change all backslashes (\) to forward slashes (/) in the path.

For example, to change the log directory to the UNC path `\\homer\service1-dir`, change the following line:

```
DSLogDir=./ds-data
```

to this:

```
DSLogDir=//homer/service1-dir
```

Renaming a Broker

Renaming a Broker can cause issues with Engine migration. Specifically, its Engines log in to a new Broker and start taking tasks, but save the old Broker name as a Broker to log in to after restart. After restart, they cannot log in to the requested Broker and restart again, repeating this cycle.

To avoid this issue, restart the Director after renaming a Broker.

Moving a Manager

If you must move a Manager from one machine to another, you must take a few things into account:

- The new Manager needs to have the same fully-qualified domain name as the old one.
- The directory structure of the new Manager needs to exactly match the old one. In other words, it needs to be at the exact same location in the file system.

Using a different IP address for the new Manager works, but other components on the grid might have cached the old IP address of the Manager. This means you might need to restart your Primary Director. If it is the Primary Director you are updating, you might need to restart all Brokers attached to it. If you are unsure whether you must restart, check if your components are logging into your Manager as expected.

GridServer Manager Administration Procedures

The following procedures are commonly used when maintaining a GridServer installation. Most of the procedures have to do with the GridServer Manager. See Managing Engines for more information about managing Engines.

Backup / Restore

Backup of GridServer Managers requires an OS-level file copy of the DS_DATA directory.

Backup Procedure

To back up a GridServer installation:

- Archive (with GNU tar or ZIP) or simply copy the DS_DATA directory. (This is the directory that is specified by `DS_DATA_DIR` variable.)

Restore Procedure

To restore a GridServer installation:

1. Unpack the original GridServer Manager installation using WinZip or a similar tool for Windows; on a UNIX system, unpack using `tar`.
2. If you previously set `DS_DATA_DIR` in the `server.bat` or `server.sh` file, edit that file in the new installation and change the definition again.
3. Copy the backup DS_DATA directory to its previous location.

Importing and Exporting Manager Configuration

GridServer Managers support the ability to export the configuration and deployment of Directors and Brokers into a signed JAR file format and later import this same format to another Manager. You can migrate Manager configurations from a UAT or prototype grid to a production grid, simplify administration of multiple Manager systems, or disseminate an organization's default Manager configuration among all clusters in the organization.

The Import/Export feature imports and exports the following component configurations:

- Manager configurations
- Service Types
- Batches/Batch schedule
- Service Runners
- Contents of the internal repository
- GridCache Schemas
- Resource files, including Grid Library files
- Credentials repository
- Server Hooks

You can only Import/Export the internal repository and Manger Configurations on Managers of the same version and update. For example, if you export a GridServer 5.1 Manager configuration, you can't import it on a GridServer 6.1 installation.

> **Note:**

> Not all Manager configuration items are exportable. Server environment-specific items, such as reporting database configuration, are not exported.

To export a configuration

1. In the GridServer Administration Tool, go to **Admin >****System Admin >****Manager Import/ Export**.
2. Select the configurations to include in the JAR. In each configuration group, you can select individual items, or choose **Select all**to include all items in that group.
3. Click **Export**.

4. A File Download dialog appears. Click **Save**to save the JAR file.

To import a configuration

1. Go to **Admin >****System Admin >****Manager Import/Export**.
2. Click the **Browse**button next to the **Provide File for import**box.
3. Browse to the location of the JAR file containing the GridServer Manager configuration export.
4. Click **Upload**to begin the import.
5. A list of configurations found in the JAR file are displayed, with configurations highlighted in red if they install over existing configurations. Select the configurations you want to import, then click **Import**.

If the Manager requires a restart for changes to take effect, a message is displayed and the Manager automatically shuts down.

Setting the SMTP Host

You can configure the GridServer Administration Tool to send email notifications, configured on the **Admin > System Admin > Email Notification**page. It also sends new account messages when a new user account is created. To send the email, you must configure a SMTP host for the Manager. This is typically configured during Manager installation, but you can add or change the value afterward.

To set the SMTP host:

1. Go to **Admin >****System Admin >****Manager Configuration > Admin**.
2. Under the **Mail**heading, in **SMTP Host**, enter the name of your SMTP server.
3. In **Contact Address**, enter the email address of an administrative contact. A notification is sent from this address to the new user when their account is created on the Administration Tool.
4. Click **Save**.

Configuring the Timeout Period for the Administration Tool

For security purposes, the GridServer Administration Tool times out after a certain period and requires users to log in again. By default, the timeout period is 60 minutes.

To change the timeout period:

1. Go to **Admin > System Admin > Manager Configuration > Security**.
2. Under the **Admin User Management**heading, change the value of **Admin Browser Timeout**.

Reconfiguring Managers when Installing a Secondary Director

When you install a Manager that includes a Secondary Director, you must also configure the Manager containing the Primary Director. This registers the Secondary Director's address with the Primary Director, as well as reconfigures the Engine and Driver configurations.

To reconfigure the Manager containing the Primary Director, select **Admin > System Admin > Manager Reinstall**, and enter the Secondary Director's address and port on the corresponding page. This configures the Primary Director to recognize the Secondary Director, and reconfigures Engine and Driver configurations accordingly.

Reconfiguring the Engine Communication Port

By default, the Manager uses port 8000 for communication with Engines. This port can be changed, but you must also change Engine configurations to log in at the new port.

> **Note:**
>
> If you reconfigure the Manager's messaging port prior to updating an Engine's configuration, the Engine is no longer able to reach the Manager. Consequently, Engines can not log in after the Manager Reconfigure process, and you must re-install them. To avoid this issue, change the port used for Engine communication first.

> To change the port used for Engine communication:

1. Go to **Grid Components > Engines > Daemon Admin**.
2. Select **Configure All Daemons**from the Global Actions list.
3. Change the Primary Director property to the URL with the new port.
4. All Engine Daemons then log off and no longer appear in the Daemons list.
5. Edit the `server.xml` file in the base directory location defined in DS_DATA. (If DS_DATA is not defined, the file is in `TIBCO_HOME\manager-data\conf` for Windows, or `TIBCO_HOME/manager-data/conf` for UNIX.)
6. To change the messaging port, replace the value of 8000 in the following line:

   ```
   <Connector port="8000"
   ```

7. Restart the Manager. After restart, all Daemons reappear in the Daemons list.

8. Go to **Grid Components > Engines > Daemon Admin**, select **Configure All Daemons**, and change all Primary Director URLs back to “default”.

Promoting a Secondary Director to Primary Director

When a Primary Director fails and a Secondary Director takes over, it has a backup copy of the internal database that is used to store user accounts and other properties. However, because it can’t resynchronize this information to the Primary Director when it returns, it is read-only and cannot be modified.

In a situation where the failed Primary Director is unavailable for some time, you might need to promote the Secondary Director to the role of Primary Director, and possibly assign another Secondary Director. Note that this requires a failover setup with two Director machines that are both capable of handling a volume necessary for your grid’s demands, plus an optional Secondary Director.

To promote a Secondary Director to Primary Director:

1. Log in to the Secondary Director in the GridServer Administration Tool.
2. Go to **Admin >****System Admin >****Manager Reinstall**.
3. Select **Manager Configuration**and click **Next**.
4. In the **Director**list, select **Primary**.
5. Finish the configuration.

Configuring SNMP

GridServer supports Simple Network Management Protocol (or SNMP), which can generate alerts (called traps) on a per-event basis. For example, you can send events such as 'Server Started' and 'Engine Died' as traps to an SNMP monitoring station. The SNMP interface is administered through an administrative plug-in on the GridServer Manager. The traps themselves are defined in a Management Interface Base (MIB), which is designed for applications. An MIB specific to DataSynapse is defined and included with the Manager.

To configure and enable SNMP support for your Manager:

1. Go to **Admin > System Admin > SNMP Event Traps**.
2. Enter the hostname and port of your SNMP server in the **Host**and **Port**fields, then click **Add**.
3. If you have multiple SNMP servers, repeat step 2 for each server.
4. In **SNMP Version**, select the version of the SNMP protocol your servers use.

5. Select each event in the event list for which you want to generate a trap.
6. Go to **Admin > System Admin > Manager Configuration > Admin**.
7. Under the **SNMP**heading, set **enabled**to `True` for the Broker, Director, or both.

The DataSynapse MIBs are at `DS_MANAGER/webapps/livecluster/WEB-INF/etc/snmp`.

Some SNMP events generate traps from the Broker, while others generate traps from the Director. The following table lists events that generate traps, sorted by Broker or Director:

SNMP Trap Events

BROKER TRAP EVENTS	DIRECTOR TRAP EVENTS
DriverAddedEvent, DriverRemovedEvent, EngineAddedEvent, EngineDiedEvent, EngineBlackListedEvent, EngineGreyListedEvent, EngineRemovedEvent, JobCancelledEvent, JobFinishedEvent, JobRunning, MemoryWarningEvent, TaskFailed	BrokerAddedEvent, BrokerRemovedEvent, EngineDaemonAddedEvent, EngineDaemonRemovedEvent, LocalDatabaseBackupFailureEvent, RemoteDatabaseBackupFailureEvent, ServerStartedEvent

LogLogic Integration

GridServer can seamlessly integrate with TIBCO LogLogic® for analysis of Manager and Engine logs. You can optionally configure GridServer to send log messages to a syslog daemon, where they can be captured by LogLogic.

Configuration

GridServer log items can be sent to a centralized syslog daemon, which is then used for capture by LogLogic. This is configured in the Manager Configuration or the Engine Configuration.

Manager Configuration

To configure syslog output for a Manager, go to **Admin > System Admin > Manager Configuration > Logging > Syslog**.

Engine Configuration

To configure Syslog output for a group of Engines, go to **Grid Components > Engines > Engine Configurations**and edit the properties in the **Logging to a Syslog Server**section. This affects all Engines using that Engine Configuration.

Configuration Properties

Configuration properties are as follows:

Syslog Configuration Properties

PROPERTY	DESCRIPTION	DEFAULT
Enabled	If syslog output is enabled.	false
Syslog Server Address	The hostname or IP address of the syslog server.	
Syslog Server Port	The port of the syslog server	514
Components	The comma-separated list of components that are logged. The component is the value that is in brackets on a log message. If the logger is set to use the Fully Qualified Class Name, the component name is just the unqualified class name. Use the * character to log all components. (The * character can only be used to indicate all, it cannot be used in component names.)	
Max Log Level	The highest level of messages that are sent to syslog. Note that log messages higher than the default log level are not sent; this setting only limits what is sent to syslog.	ALL
Tag	The syslog tag, which is typically the name of the program or process that generated the message.	GridServerEngine or GridServerManager

PROPERTY	DESCRIPTION	DEFAULT
Time Zone	The time zone of the time stamp. Currently only GMT and default time zones are supported.	
Facility	The syslog facility level.	1

Logging Message Format

Facility

The syslog facility value is an integer from 0-23. See the Syslog protocol definition in RFC 5424 for more information about facility values.

Severity

The severity is an integer from 0-7. The following table describes the severity levels, and how Java log levels are mapped to them:

Syslog Severity Levels

CODE	SEVERITY	JAVA LEVEL	DESCRIPTION	GENERAL DESCRIPTION
0	Emergency		System is unusable.	A "panic" condition usually affects multiple apps/ servers/sites. At this level it usually notifies all tech staff on call.
1	Alert		Action must be taken immediately.	Must be corrected immediately, therefore notify staff who can fix the problem. For example,

CODE	SEVERITY	JAVA LEVEL	DESCRIPTION	GENERAL DESCRIPTION
				the loss of a primary ISP connection.
2	Critical		Critical conditions.	Must be corrected immediately, but indicates failure in a secondary system. For example, the loss of a backup ISP connection.
3	Error	`SEVERE`	Error conditions.	Non-urgent failures, these must be relayed to developers or admins; each item must be resolved within a given time.
4	Warning	`WARNING`	Warning conditions.	Warning messages, not an error, but indication that an error occurs if action is not taken. For example, file system 85% full - each item must be resolved within a given time.

CODE	SEVERITY	JAVA LEVEL	DESCRIPTION	GENERAL DESCRIPTION
5	Notice		Normal but significant condition.	Events that are unusual but not error conditions - might be summarized in an email to developers or admins to spot potential problems - no immediate action required.
6	Informational	`INFO, CONFIG`	Informational messages.	Normal operational messages - might be harvested for reporting, measuring throughput, and so on - no action required.
7	Debug	`FINE,FINER, FINEST`	Debug-level messages.	Info useful to developers for debugging the application, not useful during operations.

Output

The output is in standard syslog format:

```
<PRI> HEADER MSG

PRI: Integer value: (Facility * 8) + Severity
```

```
HEADER: Timestamp Address
```

- The timestamp format is *MMM dd HH:mm:ss*
- The address can be hostname or IP. The fully qualified domain name as reported by the OS is used.

```
MSG: TAG: CONTENT
```

- The separator between `TAG` and `CONTENT` can be a number of things, but a colon is most common.
- The `CONTENT` is the GridServer log message, without the initial timestamp.

Elasticsearch Integration

To support integration with Elasticsearch, Manager log messages and Server events can be sent to a REST endpoint as JSON. This enables you to use Elasticsearch for Engine and Broker usage analysis, and to perform quicker and easier issue diagnosis with log parsing. This feature can also be used for any tool like Elasticsearch that accepts messages on an HTTP(S) REST endpoint as JSON.

Configuration

To configure REST output for logging or server events, go to **Admin > System Admin > Manager Configuration > Logging > Log to REST**or **Server Events to REST**. This feature is supported only for Managers, and not Drivers or Engines.

Log Message to REST Configuration

The following properties are related to sending log messages to a REST endpoint:

Log Message to REST Configuration

NAME	DESCRIPTION	DEFAULT
Enabled	If sending log messages to REST is enabled.	false

NAME	DESCRIPTION	DEFAULT
Server URL	The complete URL path of the location that receives the log messages. The following substitution variables are supported: • `${hostname}` the short (not fully qualified) hostname • `${managerId}` the Manager ID as found on the **Admin** section of the Manager Configuration page • `${dateFormat:` *format* `}`, where *format* specifies a date according to the Java `SimpleDateFormat` class. For example, http://logdb.mycompany.com/datasynapse-${hostname}-${managerId}-${dateFormat:yyyy-MM-dd}/log	
Username	Username, if using Basic Auth.	
Password	Password, if using Basic Auth.	
Max Log Level	The highest level of messages that is logged.	`ALL`

NAME	DESCRIPTION	DEFAULT
	This can only be used to restrict what is logged to the server from the default settings. Typically this is set to `ALL`, unless you want to send higher level messages to another logging device but restrict what is logged to this server.	
Backlog	The number of messages that can be queued for send.	1000
Discard when Backlogged	If true, subsequent messages are discarded when the backlog is reached. Otherwise, the caller is blocked.	`false`
Timestamp format	The format of the 'timestamp' property	`yyyy-MM-dd'T'HH:mm:ss.SSSZZ`
Time Zone	The time zone of the time stamp, currently only GMT and default time zone is supported.	
Excluded LogRecord Properties	A comma-delimited list of properties of the Java `LogRecord` class that is excluded. By default, the following are excluded: `sourceClassName`, `sourceMethodName` because they can impact	`millis,sequenceNumber,sourceClassN`

NAME	DESCRIPTION	DEFAULT
	performance; `millis` because a formatted timestamp is often preferred; `sequenceNumber` because it might not be useful.	
HTTP Timeout	The connect and read HTTP timeouts, in milliseconds	5000

Server Event to REST Configuration

The following table lists properties related to sending server events to a REST endpoint:

Server Event to REST Configuration

PROPERTY	DESCRIPTION	DEFAULT
Enabled	If sending server events to REST is enabled.	false
Server URL	The complete URL path of the location that receives the log messages. The following substitution variables are supported: • `${hostname}` the short (not fully qualified) hostname. • `${managerId}` the Manager ID as found on the **Admin** section of the Manager Configuration page.	

PROPERTY	DESCRIPTION	DEFAULT
	• `${dateFormat:` *format* `}`, where *format* specifies a date according to the Java `SimpleDateFormat` class. For example, http://logdb.mycompany.com/datasynapse-${hostname}-${managerId}-${dateFormat:yyyy-MM-dd}/log	
Username	Username, if using Basic Auth.	
Password	Password, if using Basic Auth.	
Backlog	The number of messages that can be queued for sending.	1000
Discard when Backlogged	If true, subsequent messages are discarded when the backlog is hit. Otherwise, the caller blocks.	false
Timestamp format	The format of the `timestamp` property	`yyyy-MM-dd'T'HH:mm:ss.SSSZZ`
Time Zone	The time zone of the time stamp. Currently only GMT and default time zone are supported.	

PROPERTY	DESCRIPTION	DEFAULT
Excluded Events	A comma-delimited list of integer event types that are excluded.	
Include Type	If true, the integer event type is included in the message.	true
Include Name	If true, the event name is included. For example, `TASK_SUBMITTED`.	false
HTTP Timeout	The connect and read HTTP timeouts in milliseconds	5000

Message Format

Each log message is JSON formatted, and includes the following information:

- `timestamp`—when the event occurred (**not**when it was sent.)
- type—The integer type, from the `ServerEvent` class in the API.
- name—The name, from the `ServerEvent` class.
- properties—The properties map for that event.

For example:

```
{
   "timestamp":"2014-04-29T17:45:18.512-0400",
   "type":28,
   "name":"TASK_SUBMITTED",
   "properties":{
      "ServiceSessionID":"21837468246",
```

```
      "ServiceInvocationID":"3".

      ...

   }

}
```

Database Maintenance

Each GridServer Manager contains an embedded database running on each Director. This internal, or admin database stores administrative data, such as User, Engine, Driver, and Broker information. You can also configure GridServer to use an external reporting database to log events and statistics. GridServer does not include a reporting database; you must use your own enterprise-grade database.

Database Types

There are two databases used by the GridServer Manager, each of which are described below.

The Reporting Database

The external reporting database is optionally used to store events and statistics. Depending on configuration settings, this database can grow quickly. Using a robust external database is recommended if you plan to make use of the reporting capabilities. The specific types of data that are stored in the reporting database are configurable in the Administration Tool at **Admin > System Admin > Manager Configuration > Database.**Install the external database on any machine reachable from the Broker over a standard network connection.

If a grid is deployed across a large WAN on which latencies can be large, it’s best not to have all Brokers write to the same reporting database, especially if the database is in a remote location. You can override a Director’s reporting database settings with a Broker-specific database setting. To do so, go to **Admin > System Admin > Manager Configuration > Database**, and set **Override Primary Director**setting to True. When True, it uses the given values, otherwise it gets the values from the Director.

For information about installing an external database for the reporting database, the *GridServer Installation Guide*.

The Internal Database

GridServer’s internal database stores admin data such as User, Engine, Driver, and Broker information. In typical cases, the internal database is read at Manager startup, and only written to thereafter if user-driven admin events occur, such as adding a user, Engine, or Manager. The internal database is required to start the Manager. This database is an embedded component of the GridServer software.

Authenticating to the Reporting Database with Active Directory (AD)

The GridServer Manager can authenticate its connection to a Microsoft SQL Server reporting database using on-premise Active Directory (AD) credentials through NTLM. This method serves as an alternative to traditional SQL login authentication.

To configure this, navigate to **Admin > System Admin > Manager Configuration > Database**. On this page, enter the details for a dedicated AD service account:

- **User Name:**Provide the bare `sAMAccountName` of the AD account (do not include the domain prefix). The domain is specified within the connection URL.
- **Password:**Enter the password associated with the AD account.

For more information about the `connection-string` parameters and SQL Server login setup, refer to "Configuring a ReportingDatabase" section in the TIBCO GridServer® Installation guide.

This setting authenticates only GridServer Manager's database connection. It operates independent of user authentication to the Manager as explained in Using Windows Authentication and Authorization and other grid component authentication to the Manager as explained in Using Pure Kerberos Authentication.

Internal Database Reset

If, for some reason, you need to reset the internal database to its original state as initially installed, you can use the following procedure.

> **Note:**
>
> This procedure removes all administrative data!
>
> To reset the internal database:

1. Shut down the Director.

2. Delete the contents of the `DS_DATA/db` directory.
3. Open the `DS_DATA/conf/installation.properties` file, and set `DSConfigureOnStartup=true`
4. Start the Director.
5. Go to the Administration Tool. You are presented with the installation page. Complete the installation, and restart.

Internal Database Backup

The internal database used by GridServer is automatically backed up at a regular interval. A copy of `Internal.properties` and `Internal.script` is made and stored in `DS_DATA/db/backup`. If a Secondary Director is installed, the internal database is replicated to the Secondary Director at the same time, and when the Secondary Director logs in.

Internal database backup is configured at **Admin > System Admin > Manager Configuration > Database**, under the **Admin Database Configuration**heading. The **Backup Cron**property dictates when backups are made by using the cron format described above. By default, it is set to back up the database at 3:00 AM daily. The `Backup` property is set to enable or disable back up. It is set to `enabled` by default.

> **Note:**
>
> Database backups can be very resource-intensive. Scheduling them during off-peak hours is recommended.

Performing Reporting Database Maintenance

GridServer Managers do not perform any cleanup of data in the reporting database. This is considered a database maintenance activity. As such, it is not managed by GridServer. Consult your database's documentation for more information about how to perform database cleanup.

To temporarily disconnect the external reporting database for periodic database maintenance, you can suspend the database connection. This backlogs all write events posted to the reporting database until reactivation.

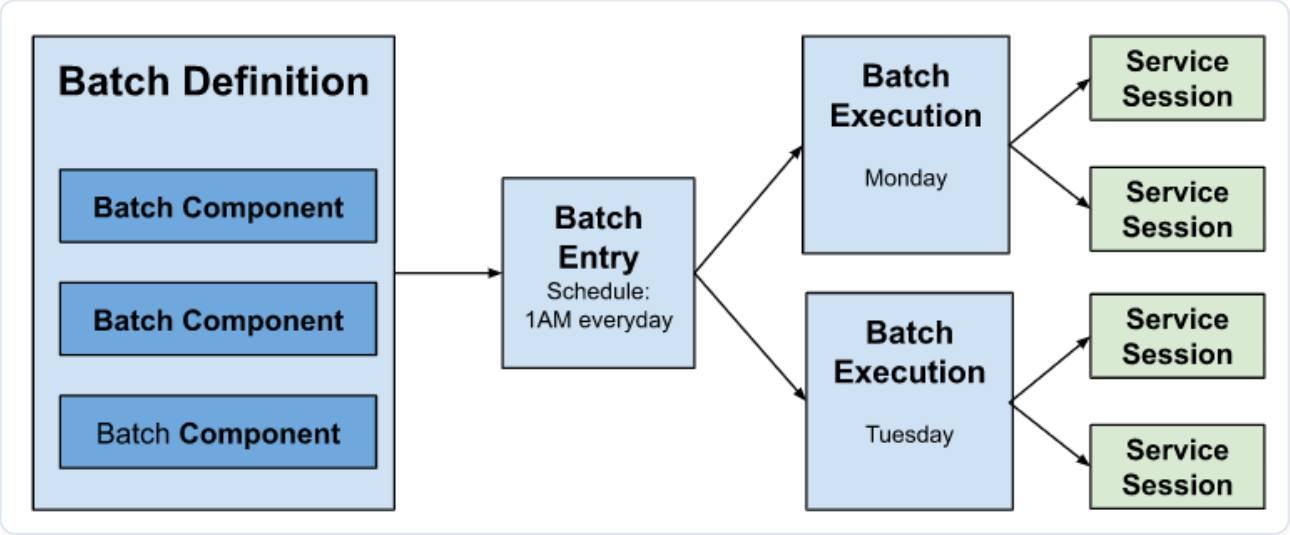
To suspend the database connection, go to **Admin > System Admin > Manager Configuration > Database**, and change the **Connection Suspended**property from False to True.

While suspended, all write events posted to the reporting database are backlogged until set back to False. You can also safely stop the database while it is suspended. Set the backlogs sufficiently high so that events are not discarded while suspended. All events are backlogged to memory.

The Batch Scheduling Facility

Commands and Services can be scheduled to run on a regular basis by using the Batch Scheduling Facility. A *Batch Definition* contains instructions in the form of components that define scheduling and which commands and services the Batch runs. When the Batch Definition is scheduled on the Manager, it creates a *Batch Entry*, which typically waits until its scheduled time, then runs, creating a *Batch Execution*. Services are run by using an embedded Driver on the Manager.

By using the Administration Tool, you can write a Batch Definition with specific scheduling instructions. You can specify a Batch Definition to immediately run when scheduled, or it can wait until a given date and time. A Batch Definition can be submitted to run at a specific absolute time, or a relative time, such as every hour. They can also be written to wait for an event, such as a new, modified, or a deleted file.



A Batch Definition consists of Batch Components. When a Batch Definition is scheduled, it creates a Batch Entry and runs as defined by the Batch Components. When it runs, it creates a Batch Execution file, which then runs the components according to the definition.

Batch Definitions contain one or more components contained within a `batch` component. A `Command` component contains a program that is run by the Batch Definition. A `schedule` or `event` component specifies when subsequent `Command` components run.

Terminology

The following terms are used to describe components related to the Batch Scheduling Facility:

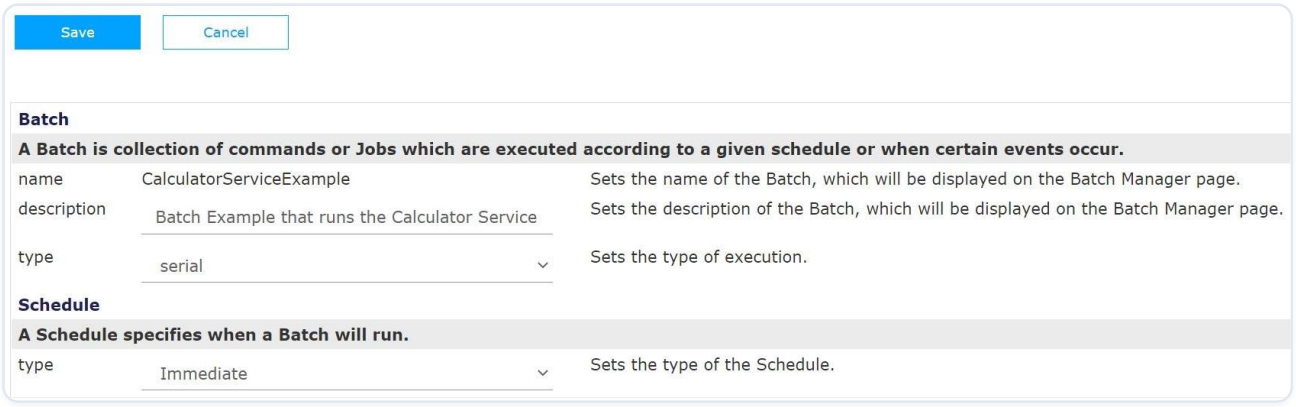
Batch Terminology

NAME	PAGE	DESCRIPTION
Batch Definition	Batch Definitions	How a Batch is written. The Batch Definition is edited from the Batch Definitions editor which contains components that define the Batch. Once created, you can manage the Batch from the **Batch Definitions**page.
Batch Component	Batch Registry editor	When a Batch Definition is created, it consists of a Batch component, which can contain other components, such as ServiceCommand components, Conditional components, and other Batch Components. The Batch Definitions editor lets you add, remove, and edit Batch components and other components it contains.
Batch Entry	Batch Schedule	When a Batch Definition has been instantiated by being scheduled on the **Batch Schedule**page, a Batch Entry is created. The Batch Entry either runs immediately, or waits to run, depending on what scheduling components are added to the Batch Definition.
Batch Execution	Batch Admin	When a Batch Entry runs, it creates a Batch Execution, which does whatever was defined in the Batch Definition. For example, if a Batch Definition uses the

NAME	PAGE	DESCRIPTION
		ServiceCommand to start ten Service Sessions, the Batch Execution does that. The Batch Execution is managed on the Batch Admin page. Any actual Service Sessions created can be managed on the **Grid Components > Services >****Service Session Admin**page.
Service Runner	Service Runners	Service Runners enable you to define a registered Service Type with options and init data that can be used in a Batch Definition.

Editing Batch Definitions

To create a Batch Definition, go to **Services > Batch > Batch Definitions**. This page contains a list of Batch Definitions on the Manager, plus a blank box for entering the name of a new Batch Definition. When you select a Batch Definition, click the **Action**list, and you can select **View/Edit Batch Definition**to edit a Batch Definition, **Rename Batch Definition**to rename a Batch Definition, **Copy Batch Definition**to copy a Batch Definition, **Delete Batch Definition**to remove a Batch Definition, **Export Batch Definition**to save an XML file of the Batch Definition, or **Schedule Batch Definition**to place a Batch Definition in the Manager's Batch queue. You can also select **Batch View**to display a graphical representation of the Batch Definition in a new window.



The Batch Definition window

To edit a Batch Definition, either select **View/Edit Batch Definition**from an existing Batch Definition's **Action**list, or type the name of a new Batch Definition in the empty box at the end of the list and click **Add**. This opens the Batch Definition window containing parameters for your new Batch Definition. You can then change the values of parameters, and click **Save**to save the values as a Batch Definition on the Manager, or click **Cancel**to exit and discard any changes you have made.

The Batch Definition parameters are as follows:

Batch Component

PARAMETER	DESCRIPTION
Name	The name of the Batch Definition. If this is a new Batch Definition, this is the name you initially typed in the blank box prior to selecting **Add**, and is not editable. (You can rename a Batch Definition by selecting the **Rename** action from the Batch Definitions page.) If an additional Batch component is added to a Batch Definition, you can set its name.
Description	Sets the description of the Batch, that appears on the Batch Manager page.
Type	Determines how a Batch Definition is run, either in serial or parallel. You can set the following types: • `parallel` - all Batch components are run when the Batch Definition is scheduled. • If set to `serial` - Batch components are run in the order in which they were added. If any of the components fail, it

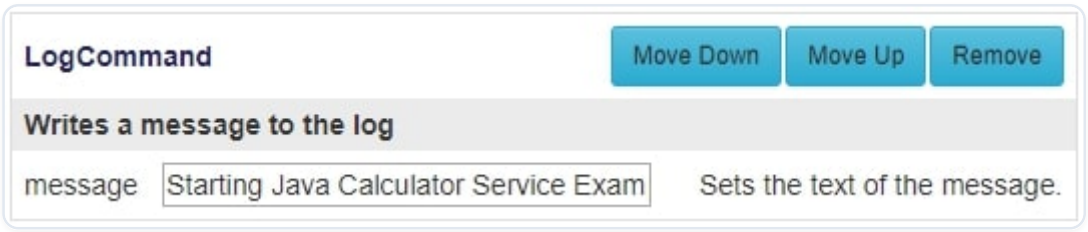
PARAMETER	DESCRIPTION
	prevents the Batch from continuing, and the Batch fails. The default is serial.

Schedule Component

PARAMETER	DESCRIPTION
Type	Sets the type of the Schedule. You can set the following types: • `Immediate` - the Batch Definition runs when scheduled. • `Absolute` - the Batch Definition runs once according to the date set in startTime. • `Relative` - the Batch Definition runs after the specified number of minutes in minuteDelay as well as repeating or running immediately with respect to repeat and runNow. • `Cron` - the Batch Definition runs according to the values set in the cron entry. (The same cron format is used as the one in the database configuration. See "Cron Format" for more information. • `Manager Startup` - the Batch Definition runs when the Manager first initializes.
Add component	Adds a component to the Batch Definition. A Batch Definition can contain one or more components.

Batch Components

The parameters in the Batch Definition editor correspond to components contained in the Batch Definition. Each Batch Definition can contain one or more Batch components. These components can be commands, events, or other Batch Definitions. For example, a LogCommand Component is shown below. To add a component to a Batch Definition, select a component from the **add component**list.



A Batch Component

Batch components are processed in a Batch Definition in order when Batch Type, described above, is set to serial. To move a component's order in the Batch Definition, click the **Move Up**or **Move Down**buttons in the upper-right corner of each Batch component. You can also remove a Batch component by clicking the **Remove**button in the upper-right corner.

Each of the types of Batch components that can be added to a Batch Definition are described below. By default, **Extended Help**is displayed. To display only the first sentence of help, select **Help**from the Help control in the upper right corner. To suppress the help display, click **No Help**.

Batch Components

NAME	DESCRIPTION
Batch	Contains another Batch Definition. This can be used to create a complex or multi-leveled Batch Definition. For example, a parent Batch Definition can start each day, starting with two child Batch Definitions, each with different schedules or conditions. For each new Batch component, you must set the same parameters for a Batch Definition as described above. You can then add additional components to the Batch.

NAME	DESCRIPTION
Conditional	Provides conditional processing when running Batches. The component specified by the **test** is run. If it runs successfully, the component specified by **success** runs. If it fails, the component specified by **failed** runs. The component specified in the **test**, returns success in the following conditions: • Command returns `Command.SUCCESS` • `ServiceCommand` creates the Service and submits the invocation without exception • `ServiceRunnerCommand` creates the Service and submits all invocations without exception
BatchReference	Contains a reference to a registered Batch Definition that gets loaded when scheduled from the Batch Registry.

Command	Runs an implemented method in a deployed class. Properties in Batch commands are based on the Java Beans specification. In a Java class, you define methods that match a pattern to get and set property values. For example, if you have a property called `MyName`, you can define two methods:
	
	`Public class MyCommand implements Command {`
	
	`public void setMyName(String value) { MyNam`
	
	`public String getMyName() { Return MyName; }`
	
	`public int run()`

NAME	DESCRIPTION

	`{`
	`; //do stuff`
	`return Command.SUCCESS;`
	`}`
	`}`
	On the Batch Registry page, adding this command to a Batch, you can add a property with the name `MyName` and value `Bob` by entering these in the two text fields and clicking **Add property**.

ServiceCommand	Starts a Service. You can specify a Service type registered on the Manager and method name to run. You can also specify a Service reference ID, Service action, and input and init data for the Service. The Service reference ID enables you to reference the Service from another Service Command. Data is comma-delimited. You can add ServiceDescription, ServiceOptions, and Discriminator components to a Service by using a Service Runner.
ServiceRunnerReference	Loads the specified registered Service Runner. See below for information about registering a Service Runner.
AdminCommand	Runs a command via the GridServer Admin API. For more information about using the

NAME	DESCRIPTION
	Admin API, see the *GridServer Developer's Guide*.
EmailCommand	Sends an email message from a Batch Definition, for notification or alerts. You can enter a comma-delimited list of email addresses for recipients, and a message string, used as a subject and a body. Note that for the batch processor to send an email, you must define an SMTP server and Contact Address in your Manager Configuration. To do this, go to **Admin > Manager Configuration > Admin**, and enter the values for **SMTP Host** and **Contact Address** under the **Mail** heading. Email sent by the batch processor is sent from the Contact Address.
EmailFilesCommand	Sends an email message from a Batch Definition that includes files as attachments. It is typically used to send the output of a previous command by saving that output to a file. You can enter a subject, a message body string, a comma-delimited list of email addresses, and a semicolon-delimited list of files, which are sent as attachments in the message. The setup rules given above in the description of the EmailCommand component also apply to the EmailFileCommand component.
ExecCommand	Runs a command from a Batch. This runs a command from the application server's root directory. You can set an input, output, and

NAME	DESCRIPTION
	error file, plus a log file for the command to be run.
LogCommand	Writes a string to the Manager log. This is useful for testing Batches or indicating when a Batch is starting or stopping.
WaitCommand	Halts for a moment before proceeding. The amount of wait time is specified in seconds. Note that this component is only useful for generating a wait time when the Batch type is serial.
EngineWeightCommand	Sets the Engine distribution weighting relative to other Brokers. Only Brokers that are part of a Broker/Director Manager installation are listed.
Event	Makes a Batch File wait for an implemented event to take place. You can use this to pause until a specific condition in a class you deployed has occurred.
FileEvent	Makes a Batch wait for a file event to occur to the specified file before completing the remaining items in the Batch Definition. Specifically, it enables you to watch a file and wait until it is created, deleted, or modified before proceeding. The file specified in FileEvent supports wildcards in the base filename. Any file matching the wild-card rule can trigger the file event. Wildcards supported include `*`, to match the rest of the base filename, and `?`, to match one character in the base filename. The wildcard naming role is the

NAME	DESCRIPTION
	same as what is implemented by `org.apache.commons.io.filefilter.WildcardFilter` (see http://jakarta.apache.org/commons/io/api-release/index.html for more information.)

Service Runners

Service Runners enable you to define a registered Service Type with options and init data that can be used in a Batch Definition. It can also be used to chain together Service Types and Discriminators into a single unit that can be used in a Batch Definition.

To create a Service Runner, go to **Services > Batch > Service Runners**. Type the name of a Service Runner in the box and click **Add**. This opens a Service Runner Editor page, where you can choose a Service Type and enter init data, a description, and method names and input data for invocations. You can also use the list at the bottom of the page to add Discriminators, Service input description data, and Service options.

The **Service Runners**page also lists all Service Runners existing on a Manager. Using the **Actions**controls, you can edit, rename, copy, delete, export, or launch each Service Runner.

Scheduling Batch Definitions

After creating a Batch Definition with the Batch Definition editor page, the Batch is listed among the other Batch Definitions on the **Batch Definition**page. However, these Batch Definitions are not actually running on the Manager yet. To create a Batch from a Batch Definition, you must first schedule it. This actually instantiates a Batch and inserts it into the Manager’s batch queue.

To schedule a Batch Definition, go to **Services > Batch > Batch Definitions**, and find the Batch Definition in the list. Select **Schedule Batch Definition**from the **Actions**control. This schedules the Batch Definition and opens the **Batch Schedule**page, displaying the new Batch Entry.

Note that when you schedule a Batch to run at some point in the future, and then you edit or change Batch Definitions, the Batch uses the Batch Definitions as they were defined when the Batch was scheduled, and not the current ones. If you want to change a Batch Definition for a Batch you have scheduled in the future, you must remove the scheduled Batch, change the Batch Definition, and reschedule it.

The Batch Schedule Page

Batch Entries on a Manager can be listed and administered on the **Batch Schedule**page. To do this, go to **Services > Batch > Batch Schedule**. All Batch Entries resident on the Manager are listed. To remove or edit an existing Batch Entry or view logs or Batch executions, select a command from the **Actions**control next to the relevant Batch.

Running Batches

Batch Entries automatically run when they reach the scheduled time or conditions defined in their Batch Definition. When this happens, Batch Executions are created and displayed on the **Services > Batch > Batch Admin**page. PDriver Batches (which are also Batch Executions) are also displayed on this page. From this page, you can monitor Batch Executions, search for logs, and display the Batch Monitor to view what parts of a Batch have completed.

Any Services that are run by the Batch Execution are displayed on the **Services > Services > Service Session Admin**page. From there, you can cancel Service Sessions, view tasks, or do any other actions you normally do with a Service. Note that it is possible to have a Batch Execution run a Service that continues to run, even after the Batch Execution reports that it is finished.

Deploying Batch Resources

Java Services, Commands, and other resources must be placed in `DS_DATA/batch/jar` to be properly loaded by the embedded Driver.

Batch Fault-Tolerance

Batch Schedules that exist on a Manager are persistent, provided the **Next Run**field is not `never`. This provides failover capability in the event of a Manager failure, as the Batch Schedules still exist when the Manager is restarted.

The following Batch Schedules are persistent:

- Absolute schedules
- Relative schedules with repeat
- Cron schedules

All persistent Batches are restarted when the Manager is restarted, as if they are scheduled for the first time. Batch runs that were to occur during the time when the Manager is down are ignored.

Optimizing the Grid

This section provides information about tuning your Grid's settings for optimal performance.

Diagnosing Performance Issues

To find bottlenecks in application performance, use GridServer's Instrumentation feature. With instrumentation enabled, you can get detailed timings of each request submitted to the Broker. These timings highlight scheduling overhead, data marshaling time and network delays.

Note that Instrumentation measures only GridServer-related times. It does not show other application delays due to, for example, excessive database load.

For information about turning on Instrumentation, see Enabling Enhanced Task Instrumentation. Also see the *GridServer Developer's Guide*.

Tuning Data Movement

Efficient handling of data can often make or break achieving performance gains in a Grid-enabled application. Instrumentation reveals problems with having too much data per request: serialization, deserialization and network transport times are high compared to the actual Engine-side compute time. There are a number of remedies for inefficient data movement. We survey them here in order from simplest to most complex.

Set Invocations Per Message > 1

By default, the Driver sends one message to the Broker per task submission. For Services that use the model of 'Submit Many, Collect Results", this is not efficient. Any Service that submits a reasonably large number of tasks at one time must always have the `InvocationsPerMessage` option set greater than one. (The optimal number depends on the environment; 25-50 is a good starting point). After submission, `flush` must be called on the Service to send any remaining buffered tasks. (`waitUntilInactive` and `destroyWhenInactive` also implicitly flush.)

Collect After Submit

The default mode of results collection is immediate. As in the previous section, if your model submits many tasks, then waits for results, this can cause inefficiencies. Specifically, the collection of results can slow down submission of tasks, resulting in idle Engines that can be working on your Service. Setting the collection type to `AFTER_SUBMIT` ensures that the tasks are worked on as soon as possible. After submission, call `waitUntilInactive` or `destroyWhenInactive` to begin collection.

Stateful Processing

GridServer supports two related mechanisms that link client-side service instances to Engine-side state, thereby reducing the need to transmit the same data many times. The two mechanisms are initialization/update data, and Service affinity.

Initialization/Update Data

Making data that is constant across an entire set of task requests into Service initialization data is recommended. Initialization data is transmitted once per Engine, rather than once per request. Designing long-lived volume-based applications that typically process thousands of requests, and compute-intensive applications so that they create many small requests, rather than few large ones is also recommended, for a variety of reasons. See the *GridServer Developer's Guide* for more information.

If a piece of data is not constant throughout the life of the application, but changes rarely (relative to the frequency of requests), it can be passed as initialization data and then changed by using an update method.

The **Service Session Size**parameter, located on the **Grid Components > Engines > Engine Configurations**page under the **Service Caches**heading, controls how much initialization data can be stored on an Engine in aggregate. In other words, if the total size of init data across all loaded service instances exceeds the set value of the parameter, then the least-recently used Service instances are purged from the cache. If Instrumentation shows a non-zero time for Engine Download Instance the second or subsequent time an Engine receives a request from a service, that indicates that the service instance was purged from the cache. Increasing **Service Session Size**might then result in improved performance.

Affinity

The GridServer scheduler uses the fact that an Engine has initialization data and updates from a particular Service to route subsequent requests to that Service. This feature, called affinity, further reduces data movement, because unneeded Engines are not recruited into the Service. (However, if the Service has pending requests, available but uninitialized Engines are allocated to it.) Affinity can be further exploited by dividing the state of an application across multiple client-side Service instances, called Service Sessions. The application then routes requests to the instance with the appropriate data. For example, in an application dealing with bonds, each Service instance can be initialized with the data from one or several bonds. When a request comes in for the value of a particular bond, it is routed to the service instance responsible for that bond. In this way, a request

is likely to arrive on an Engine that already has the bond data loaded, yet no Engine is burdened with the entire universe of bonds.

The `STATE_AFFINITY` Service option is a number that controls how strongly the scheduler uses affinity for this service. The default is 1, so set it to a higher value to give your service preference when Engines are being allocated by affinity.

The `AFFINITY_WAIT` Service option controls how long a queued request avoids allocation to an available Engine that has no affinity, in the hope of later being matched to an Engine with affinity. Use this option when the initialization time for a service instance is large. For instance, say it takes five minutes to load a bond. If `AFFINITY_WAIT` is set to two minutes, then a queued request is not assigned to an available Engine that lacks affinity for two minutes from the time the first Engine becomes available. If an Engine that already has loaded the bond becomes available in those two minutes, then the request is assigned to that Engine, saving five minutes of startup time.

The `AFFINITY_DEPTH` Service option is used for invocation-level affinity to determine how deep into the request queue the affinity score must be calculated between all available Engines. It must be greater than zero (the default) if adding affinity to tasks. Larger values can result in longer scheduling episodes, so this number must be chosen wisely.

Affinity can also be set based on Engine Properties instead of state, by using a *Property Affinity Condition*. The scheduler then calculates the affinity score based on state, and then adds a defined number for each satisfied Property Affinity Conditions you have added to the Service.

Compression

Setting the `COMPRESS_DATA` Service option to true (in the Service client or on the **Services > Services > Service Type**page) causes all transmitted data to be compressed. For large amounts of data, the transmission time saved more than makes up for the time to do the compression.

Packing

Packing multiple requests into a single one can improve performance by amortizing the fixed per-request overhead of GridServer and the application over multiple units of work. The fixed overhead includes TCP/IP connection setups for multiple transits, GridServer scheduling, and other possible application initialization steps.

GridServer's `AUTO_PACK_NUM` Service option is an easy way to achieve request packing. If its value is greater than zero, then that many requests are packed into a single request, and responses are unpacked, transparently to the application. (If the application makes fewer than

`AUTO_PACK_NUM` requests, then the accumulated requests are transmitted after one second.) Auto-packing amortizes per-request overhead, but does not factor out common data.

Direct Data Transfer

By default, GridServer uses Direct Data Transfer (DDT) to transfer inputs and outputs between Drivers and Engines. When Driver-Engine DDT is enabled, the Driver saves each request as a file and sends a URL to the Broker. The Engine assigned to the request gets the URL from the Broker and reads the data directly from the Driver. Engine-Driver DDT works the same way in the opposite direction. Without DDT, all data must needlessly go through the Broker.

DDT is efficient for medium to large amounts of data, and prevents the Broker from becoming a bottleneck. However, if the amount of data read and written is small, disabling DDT might boost performance.

Disable Driver-Engine DDT in the `driver.properties` file on the client. Disable Engine-Driver DDT from the **Grid Components > Engines > Engine Configurations**page.

Shared Directories and DDT

In some network configurations, it might be more optimal to use a shared directory for DDT rather than the internal file servers included in the Drivers and Engines. In this case, the Driver and Engines are configured to read and write requests and results to the same shared network directory, rather than transferring data over HTTP. All Engines and the Driver must have read and write permissions on this directory. Shared directories are configured at the Service level with the `SHARED_UNIX_DIR` and `SHARED_WIN_DIR` options. If using both Windows and UNIX Engines and Drivers, you must configure both options to be directories that resolve to the same directory location for the respective operating systems.

Caching

Service initialization data is effectively a caching mechanism for data whose lifetime corresponds to the Service Session. Other caching mechanisms can be used for data with other lifetimes.

If the data is constant or rarely changing, use GridServer's resource deployment mechanism to distribute it to Engine disks before the computation begins. This is the most efficient form of data transfer, because the transfer occurs before the application starts.

GridCache can also be used to cache data. GridCache data is stored on the Manager and cached by Engines and other clients. See the *GridServer Developer's Guide* for more information.

Data References

GridServer supports Data References: remote pointers to data. A Data Reference is small, but can refer to an arbitrary amount of data on another machine. Data References are helpful in reducing the number of network hops a piece of data needs to make. For instance, imagine that an Engine has computed a result that another Engine might want to use. It can write this result to GridCache. But if the result is large, it travels from the writing Engine to the GridCache repository on the Broker, and then to the reading Engine. If the first Engine writes a Data Reference instead, the second Engine can read the data directly from the first Engine. Data References hide this implementation from the programmer, making network programming much simpler.

The data referenced in a data reference is periodically deleted. By default, this happens every 168 hours, or 7 days. You can configure this time, either to retain data for a longer period, or to delete data more frequently and free space on the client and Engine filesystems. To change this period, go to **Admin > System Admin > Manager Configuration > Services**. Under the **Data Transfer**heading, change the value of **File Time To Live**, and click **Save**.

HTTP Proxy for Engine Data Transfer

In a GridServer deployment where a Broker and its Engines are separated by a WAN, it can be inefficient to transfer the same data over the WAN to multiple Engines from the Broker or the Clients. One solution is to use an HTTP proxy server (such as Squid Web Cache) to cache the session's init data, which any Engine that works on the session must transfer. You can specify a proxy server in an Engine configuration, and the proxy server caches the Service data for other Engines also using the same proxy server.

For more information about using an HTTP proxy for Engine data transfers, see Configuring a Caching HTTP Proxy Server

Diagnosing GridServer Issues

This section contains information about how to find information to diagnose GridServer issues. It contains information about troubleshooting your installation and gathering information that is helpful if you contact TIBCO for support.

Troubleshooting Overview

When troubleshooting a GridServer installation, first try the following:

1. Ensure that there are no service outages in your environment, such as network or database changes.

2. Read the log files, as described below.
3. Go through the relevant diagnostic sections below.

Reporting an Issue

To report a GridServer issue, go to the TIBCO Customer Support site at support.tibco.com.

Obtaining Log Files

There are several logs generated by GridServer. Depending on what kind of issue you are troubleshooting, you might need to examine one or more logs. These include Manager, Driver, Engine, and Engine Daemon logs.

The current log is always named `gridserver.log`, `driver.log`, or `engine-Instance#.TIMESTAMP`. When logs reach a configured size, they are “rolled over”, and the old log is time stamped. Logs are also moved when a non-Driver component first starts, and are timestamped with the last modification date of the file. Optionally, the process ID of the component doing the logging can be added to the filenames of the current log file.

Manager Logs

Manager Logs are generated on the console window on Windows machines if the Manager is not run as a service. They are also generated on UNIX machines if the Manager is run in the foreground on the console. Because GridServer is usually run as a service or in the background, there are several other ways to view the Manager log:

- In the GridServer Administration Tool, go to **Diagnostics**> **Real Time Log**. This displays new lines of the log as they happen, in a new window. However, it does not display any historical information. Click the **Snapshot**button to open a frozen duplicate of the current log window.
- Go to **Diagnostics > Manager Diagnostics**. This page enables you to display the Manager log, and other information in a specified time period. You can display or create a .ZIP file of the results.

  For example, to view Manager log results, select **Manager Log**, then select a time range. You can then display the log on-screen by clicking **Display**, or save it in a compressed file by clicking **Download**.
- The Manager log is available directly at `DS_DATA/logs/server/*` or the location specified at **Admin > System Admin > Manager Configuration > Logging**.

The Manager log can be set to different levels of granularity, ranging from **Severe**, which provides the least amount of logging information, to **Finest**, which logs the most information. By default, this level is set at **Info**. For debugging purposes, it might be necessary to set the level higher, to **Finer**or **Finest**.

To change the log level:

1. Go to **Admin > System Admin > Manager Configuration > Logging**.
2. In **Default Debug Level**, select a new level.

Configuration options for Manager logs, such as maximum file length and timestamp format, can also be set on the **Manager Configuration**page, in the **Logging**section.

> **Note:**
>
> If you set the log level to Finer or Finest, a large volume of logs are generated, which can adversely affect performance and overwhelm your Manager's file system over time. Remember to change back the log level after you are done troubleshooting.

Engine and Daemon Logs

Each Engine and Engine Daemon generate their own logs. These logs can be accessed directly on Engines. However, because Engines are typically installed in several different machines, there are ways to view logs remotely from other computers. The following procedures describe how to read Engine logs.

To view log files directly, look in the following directories in each Engine install directory:

- Engine instance logs: *install-dir*`/work/`*machine name*`-`*instance*`/log/*`
- Engine Daemon logs: *install-dir*`/profiles/`*machine name*`/logs/engined.log`
- Also examine other `.log` files in the Engine tree.

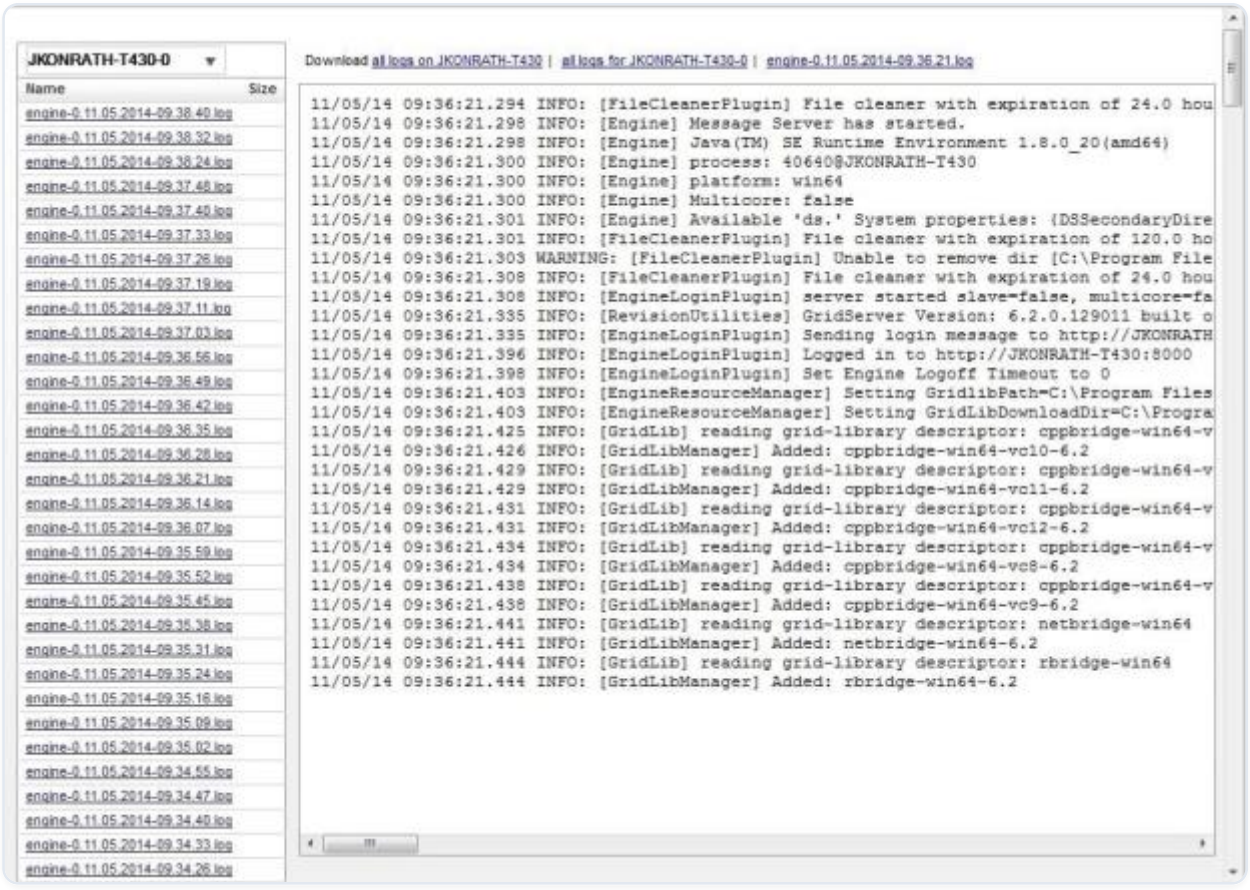
To read a remote Engine log in a scrolling window

1. Go to **Grid Components > Engines > Engine Admin**.
2. Select an Engine.
3. From the **Actions**list, select **Remote Log**.

This opens a window that displays the log for the Engine. Logging information is displayed as it is generated. This does not, however, display any prior logging history.

To access previous logs remotely

1. Go to **Engines > Engine Admin**.
2. Select an Engine.
3. From the **Actions**list, select **Log Files**.
4. A window opens with a list of links for each of the logs residing on that Engine. The links are displayed by date and time. You can do any one of the following actions:
   - Select an Engine Daemon or a particular Engine from the list in the upper left. This shows all of the log files on that Engine Daemon or Engine and their sizes. You can also type in the list box to filter the list to partial matches quickly.
   - Click a log file name and its content is displayed to the right in the window.
   - Click the links in the upper right to download a ZIP archive of all log files on the host, a ZIP archive of all log files of an Engine Daemon or Engine instance, or a particular log file.



The Log Files window

You change the log level for Engines in Engine configurations, in the **Log**section. You can also set other logging options in Engine configurations, such as the maximum log size, and when log files are automatically cleaned.

Application Server Logs

The Tomcat application server used to run the GridServer Manager also generates logs that can be helpful in diagnosing issues. Logs are maintained in the base directory in the `logs` directory.

Monitoring the Tomcat Application Server

When troubleshooting the Tomcat Application Server, you can use JMX to monitor Tomcat.

For more information about using JMX and Tomcat, see http://tomcat.apache.org/tomcat-6.0-doc/monitoring.html.

Monitoring Engines Using JMX

JMX can also be used to monitor Engines' JVMs. You can enable this in an Engine Configuration, in the **Engine JVM**section, by setting a command-line argument to start the JMX server.

A parametrized incremental port is available for passing into Engine JVM arguments. For example, setting the command-line argument `-Dcom.sun.management.jmxremote.port=${startport:9977}` sets the JMX port to 9977 plus the Engine instance number.

Diagnosing Network Issues

A common issue in a GridServer grid is that a misconfiguration in component communication on a network causes Service failure. This section gives a few common issues to troubleshoot in this situation. For more information about configuring GridServer with regard to networks, see the *GridServer Installation Guide*.

Direct Data Transfer (DDT)

By default, DDT is enabled. When a Service creation or request is initiated on a Driver, the init data or request argument is kept on the Driver, and only a URL is sent to the Manager. When an Engine receives the request, it is sent to the URL and downloads the data directly from the Driver. DDT is set in the `driver.properties`, by setting `DSDirectDataTransfer` to true. Also, by default, the Driver downloads the output data directly from the Engine. This is set in the Engine Configuration by setting the **Direct Data Transfer Enabled**option to `true`. The Driver uses an internal file server by default, at the next available port from the value of `DSWebserverPort`, set in the

`driver.properties` file. The Engine listens, by default, on port 27159. This is set in the Engine Configuration.

DDT transfers input and output data over HTTP using the Apache HttpClient library. This transfer activity is logged at the `Finer` level in both the Engine and Driver logs. It does not display at the `Fine` level. To verify that the data is being transferred directly, set the Engine and Driver log level to `Finer`.

When the Engine reads the input data, the download of the input (`.in`) file displays in the Engine log. For example:

```
Finer: [MainClientExec] Executing request GET /ds-7466344146886677638/5.in
HTTP/1.1
```

When the task retrieves the output data, the download of the output (`.out`) file displays in the Driver log. For example:

```
Finer: [MainClientExec] Executing request GET /data/bapa101-0/ddt/
ds-2491378560399007707/0.out HTTP/1.1
```

The remote host and port for each request, (for example, `192.168.32.137:1420` for the Driver's file server, or `engine-host:27159` for the Engine) display in the adjacent Opening connection and Host: header lines.

The full DDT URLs (of class `com.livecluster.util.UrlByteSource`) are also visible in the task and result data in the Driver log. For example:

```
Finer: [wire] http-outgoing-0 >> "<dataSource
c="com.livecluster.util.UrlByteSource" ...><property n="url" v="<http://
192.168.32.137:1420/ds-7466344146886677638/5.in>"/>"



Finer: [wire] http-outgoing-0 << "<resultSource
c="com.livecluster.util.UrlByteSource" ...><property n="url" v="<http://
10.126.209.12:27159/data/bapa101-0/ddt/ds-2491378560399007707/0.out>"/>"
```

The most common problems with DDT are firewall issues. Use `telnet machine:port` to test connections between components. If you are having problems with DDT, try disabling it and temporarily running in data transfer mode.

> **Note:**NET 4.8 and .NET 8 Driver require `FINEST` (not `FINER`) on the driver side — `HttpSupport.Get`() gets the URL log on `Level.FINEST`.

Data Transfer

Setting the `DSDirectDataTransfer` option in `driver.properties` to false causes the Driver to upload all input data to the Broker. Also set **Direct Data Transfer Enabled**to `false` in your Engine configurations, and the Driver downloads the output data from the Broker. The data transfer settings for the Broker are at **Admin > System Admin > Manager Configuration > Services**, under the **Data Transfer**heading. They are **Store Input to Disk**and **Store Output to Disk**.

Connection Issues

When using data transfer mode, messages like the following message mean that the Engine has lost connection to the Director:

```
Warning: [WebServerBridgePlugin] Failed ping attempt on http://
161.2.27.160:27159/data/ping.html, java.lang.RuntimeException:
java.net.ConnectException: Connection refused: connect
```

Ensure that the Director is running and that the IP address in the log is the IP address of your Director. From your Engine machine, telnet to port 27159 of the Director. If the connection is refused, you might have a network problem. When using DDT, telnet between Driver and Engine to rule out network problems.

Timeout Issues

If you get timeout messages such as the following message, you might need to adjust the configuration for this client:

```
04/15/05 09:18:57.964 Warning: [global] Error reading from http://
10.47.117.158:27159/data//2500-dklptt3z-0/ddt/4937299722762820807 /
0.out.z: java.io.IOException: Timed out reading

http://10.47.117.158:27159/data//2500-dklptt3z-0/ddt/
4937299722762820807/0.out.z
```

With Direct Data Transfer (DDT), the settings to adjust are at **Admin > System Admin > Manager Configuration > Communication**, under the **Data Transfer**heading. If the Driver is timing out attempting to read the output file from the Engine, increase the values under **Driver Data Transfer**. Also check that you can access the Engine's file server port 27159 using telnet.

If you are not using DDT for Engines, or if you are using the .NET Driver, the relevant settings are under the **HTTP Connections**heading of the page above. For example, you might want to increase the **Read Timeout**setting.

Diagnosing Engine Issues

The following section gives some information about reading logs related to Engines, and solutions to some common issues.

If problems occur with one particular Engine on the grid and the cause is not immediately obvious, it might be easier to reinstall the Engine rather than go on a long troubleshooting exercise. If the problem persists after reinstallation of the Engine, then investigate for issues with the network, application, or machine setup.

For information about managing Engines, see Managing Engines

Engine Logins, Restarts, and Failures

After GridServer starts, Engine Daemons and Engines log in to the Manager. In the Manager log, messages similar to the following message appear when this happens:

```
Info: [EngineEvent] EngineDaemon:S08048-10.103.8.48:Added

Info: [EngineEvent] Engine:Joe-0:Added
```

The Broker sends periodic heartbeats to the Engine. If these heartbeats fail, you see messages similar to the following message:

```
Info: [ProxyMonitorPlugin] Killing proxy S08049-10.103.8.49 on
EngineDaemonServicePlugin

Info: [EngineEvent] Engine:S08048-0:Logoff:Killed by the proxy monitor

Warning: [EmploymentOfficePlugin] Engine:S08048-0:Died
```

If the Engine cannot perform a heartbeat with the Broker then after 3 retries, you see the following message:

```
Warning: HeartbeatPlugin Couldn't send a heart beat to the Manager failure
to process HTTP request in POST: Connect failed, so the client logs off.
```

If the Engine Daemon fails, you see the following message:

```
Warning: [EngineDaemonServicePlugin] Engine Daemon:S08049-10.103.8.49:Died
```

You can lengthen the period between heartbeats at **Admin > System Admin > Manager Configuration > Communication**.

If the Engine fails, by default it restarts and tries to log in again. Failure messages in the Manager log look similar to the following message:

```
Info: [Scheduler] Engine:NCSILS9027B1GRD-0:Logoff:Ping failed on local
webserver, restarting instance in one minute

Fine: [EngineProxy] Logging off: NCSILS9027B1GRD-0

Fine: [EngineLoginManagerPlugin] Logging off proxy + NCSILS9027B1GRD-0
code=3

reason=Ping failed on local webserver, restarting instance in one minute

Info: [EngineEvent] Engine:NCSILS9027B1GRD-0:Removed
```

The following information is included in the Engine Daemon log reports:

```
Info: [Scheduler]
Engine:nldn8347dww-0:NotifyKillTask:120811924537238831 3-120811924537238831
3-0

Info: [Scheduler]
Engine:nldn8347dww-0:TaskDied:120811924537238831 3-120811924537238831 3-0

Info: [Scheduler] Engine:nldn8347dww-0:Logoff:Killed by the proxy monitor

Warning: [EngineEvent] Engine:nldn8347dww-0:Died
```

```
Info: [EngineEvent] Engine:nldn8347dww-0:Removed
```

JVM Issues

If there are messages similar to the following message in the Manager log, the JVM might be running out of memory:

```
Severe: [HeartbeatPlugin] while sending heartbeat
java.lang.OutOfMemoryError: unable to create new native thread
```

You can increase the Engine JVM maximum heap size in the Engine configurations.

If an Engine fails, or the logs on an Engine end abruptly, the cause might be a Java failure. Check for Java HotSpot compiler error logs in the Engine root directory. They have names like `hs_err_pidXXXX.log`, and contain information about problems in native code. The information can be used for a web search to see if it is a known problem. Also check if any native C code is being called by the application that fails.

Connection and Firewall Problems

A common problem is that Drivers or Engines are not connecting to the Director or Broker. This is typically due to Firewall or DNS issues. Correct DNS configuration is essential in GridServer installations. Use the `telnet` command to test connections from the Manager to the Engine and conversely.

All supported Windows versions enable the Windows Firewall by default. This automatically blocks any incoming traffic. To make sure that your Engine can properly communicate, the inbound port for the Engine's File Server must be open to traffic. By default, this port is set to 27159. It can be changed in the Engine Configuration. Configure your Windows Firewall to enable use of this port by your Driver machines.

Another possibility is that one of the components is assigning ephemeral ports outside of the range that can be opened. Sometimes systems assign ports outside the range of 49152-65535. You can check this by using `netstat -a`.

Engine Daemon Cannot Log On to Manager

If an Engine Daemon does not connect to a Manager, check the URL in the `intranet.dat` file in the root of the Engine installation directory and see if you can make a connection to it from the Engine machine.

Thread Dumps on Engines

To get thread dumps on Engines, use the java Visual VM tool. It is available at https://visualvm.github.io/.

Using Fusion to Debug .NET Assembly Load Failures

With C# code, a runtime library load failure can take a number of forms such as a `FileLoadException` and might be difficult to debug. The notification only mentions that the assembly load failed, but not why it failed.

To obtain more detailed debugging information about assembly load/bind failures, use the Microsoft Fusion logging system. It is included in Visual Studio .NET:

1. Start `FUSLOGVW.EXE` before opening your application.
2. Open your application.
3. After the failure has occurred, click the **Refresh**button in the Fusion logging window. An entry related to the process you just ran must appear.
4. Highlight this entry and click **View Log**to get a detailed report of the .NET Framework's attempts to load your assemblies.

Diagnosing Driver Issues

The following section gives some information about reading logs related to Drivers, and solutions to some common issues.

Driver Cannot Log In to Manager

When a Driver cannot log in to a Manager, messages similar to the following message appear in the Manager log:

```
LoginPlugin Can't log in to PrivateBrokerConnection

LoginPlugin Failed login on http://162.60.27.152:8000/livecluster/
director/PublicDriverChannel so trying next. Error:

File not found: http://162.60.27.152:8000/livecluster/director/
PublicDriverChannel
```

When this happens, try the following:

- Ping and telnet to the Manager and port number to test network connectivity.
- Check the username, password, and Director values in the `driver.properties` file or values coded in application.
- If using SSL, make sure the certificate is valid.

Client Timeout Issues

You might see timeout error messages in the Driver log similar to the following message:

```
java.io.IOException: Unexpected exception while reading data from http://
172.24.68.49:1667/ds-6353115010724789381/job.tasklet:
com.livecluster.util.threadpool.TimeoutException: Thread DefaultPool-6:

http://172.24.68.49:1667/ds-6353115010724 789381/job.tasklet timed out
```

The **Client Timeout**setting allows for the Driver to log back in after logging off due to temporary network issues, without interrupting Services.

Go to **Admin > System Admin > Manager Configuration > Communication**. Under the **HTTP Connections**heading, you can increase read and write timeout values for Drivers.

Also, the Driver must not fail over to another Broker too quickly. This is set in `DSBrokerTimeout` in the `driver.properties` file. The default is 300 seconds, or 5 minutes.

To ensure your tasks are not lost in the event of Broker failure, perform the following actiond:

1. GNavigate to **Admin > System Admin > Manager Configuration > Engines and Clients**.
2. Under the **Engine Login**heading for a period of time longer than the **Client Timeout Minutes**on the same page, under the **Client Management**heading, set **Engine Timeout Minutes**at. For example, if **Client Timeout Minutes**is set to 5 minutes, a good number for **Engine Timeout Minutes**is 8 minutes. Note that changes to this value are only applied at the next Engine log in.

Manager Turning Away Clients

By default, Drivers are turned away if there is a version mismatch between Client and Manager. A message similar to the following message appears in the Manager log:

```
[BrokerLoginManager] Turning away client: Client module version mismatch:
ClientJavaVersion version: 7.0.0: should be 7.1.0.
```

To override this, go to **Admin > System Admin > Manager Configuration > Admin**, and under the **Version Management**heading, set **Allow Driver Version Mismatch**to `true` until the Client can be upgraded.

Diagnosing Manager Issues

The following section gives some information about reading logs related to Managers, and solutions to some common issues.

Manager Port Issues

If your Broker does not log in to the Director, check the Manager logs for a message similar to the following message:

```
BindException while attempting to create a ServerSocket at myaddress:5635
```

This means another process is using port 5635, which is used for communication between the Broker and Director. If it is another Director, or another process that cannot be stopped, that port number must be changed on the Director and Brokers to another port. Use the `netstat –a` command to check port usage.

Out of Memory Issues

If your Manager appears to run out of memory, and an out of memory error is printed to the Manager log, you might need to increase the Java heap size. This typically only happens if the Manager is running many Services with many inputs.

The Java maximum heap size is set in the `server.sh` or `server.bat` file, and is 4096 MB by default. It can be increased by changing the environment variable `MAX_HEAP`.

Deployment Issues

The GridServer Manager is responsible for deploying application resources to Engines. The Manager periodically generates a checksum list of every file in its resources area. If any checksums have changed, or have been deleted or added since the last time the list was generated, it notifies the Engines.

Do not deploy resource files that only differ by case in UNIX. Resource files names must be different. If you deploy different files named `hello.zip` and `Hello.zip`, they collide on Windows machines. Also, you cannot deploy files that contain `#` or end in `.tmp`.

For more information about resource deployment and Grid Libraries, see Deploying Services.

Grid Library Issues

If you have a `GridLibraryException` failure when deploying a Grid Library, check your `grid-library.xml` for incorrect file names. Also ensure that files and file reference in `grid-library.xml` are of the correct case. Also check that the `jar-path`, `lib-path`, `assembly-path`, and `command-path` are correctly specified.

.Net GridLib - DLL Missing

When deploying a .NET assembly, all DLL libraries in `win32/lib` and all other directories specified in Engine Configuration are not used. Put all related DLL files into the same Grid Library, or another Grid Library that this Grid Library depends on. A missing DLL causes a log message similar to the following:

```
INFO  DSForward Message DD7C5440-D3B9-42D6-9201-F7E7CE3AE847 submitted

DSHandler Service invocation 0 failed for DD7C5440-D3B9-42D6-9201-
F7E7CE3AE847

Exception: DataSynapse.GridServer.Client.ServiceInvocationException

Message: spri12d10052

Source: GridServerNETClient
```

Service Failures

If a Service is not defined in the **Services > Services > Service Types**page, a message similar to the following is produced in the Driver log:

```
Service Type not found in the Service Type Registry: JavaCalculator
```

If the JAR file containing the class of the Service has not been loaded into the correct library, a message in the Driver log is produced:

```
Info: [TaskDispatcher$DispatcherJob] Canceling due to task failure and
AutoCancel.LIBRARY_LOAD, Exception type: java.lang.ClassNotFoundException
```

If a Service fails, an error is returned to the Callback method. Check your code for errors.

```
Info: [ServiceEvent]
SubmittedTask:3393405147689298243-3393405147689298243-1

CalculatorCallback::handleResponse : Result for task [B@c713d2

Info: [ServiceEvent]
CompletedTask:3393405147689298243-3393405147689298243-0:Total:2

CalculatorCallback ::handleError : Id = 1, Exception :
java.lang.Exception: service failed [java.lang.Exception: service failed
```

If a Service fails to run even if there are available Engines, check that the Service or Engine discrimination is not preventing the Service task from running. Also check for usage of dependencies in the application. The Service or task might be waiting for a Service or task it is dependent on.

GridCache Issues

GridCache is a repository on the Broker that is cached by Drivers and Engines (This is the GridCache Global mode). The Driver or Engine writes to the GridCache and then the Broker caches them automatically to the Server and propagates the changes so the file appears local to Drivers and Engines. When the cache is changed, the Manager sends invalidation messages in the heartbeat messages to the Engines and Drivers.

For more information about GridCache, see the *GridServer Developer's Guide*.

Timeout Issues

Messages similar to the following message are caused by timeouts accessing GridCache:

```
com.livecluster.util.threadpool.TimeoutException: Thread DefaultPool-91:
sun.net.www.protocol.http.HttpURLConnection:http://chialseg45:8000/
livecluster/gridcache/?query=1/
```

You can increase GridCache client read and write timeout at **Admin > System Admin > Manager Configuration > Cache**.

Eviction of Cache Entries

There is an eviction process, which removes entries from the cache when they expire. A message similar to the following message appears in the logs:

```
Finer: [GridCache] Evicting stale entries from the cache
```

A `Time To Live` value is configured in the cache schema. The default is -1, meaning entries never expire. In the case of a Global Cache, if an entry has expired, it is removed, and hence removed from all of the locally cached copies. Note that when a schema changes, the changes only apply to regions created after that schema change. The changes do not affect existing regions using the schema.

In addition, there is also a Keep Alive time associated with the cache, set in the cache schema. This specifies how long the locally cached copies of the region remain on the client once there are no longer any local references to the cache. References are decremented when you call `close()` on the cache or the Cache reference is collected by garbage. The default is 60 seconds.

Client Runs Out of GridCache Space

If the in-memory cache size is exceeded, information is pushed to disk (except for .NET Drivers or CPP Drivers, which do not support disk-based cache.) If the disk cache size is exceeded, older entries from the disk cache are removed to make way for new entries.

For example, if the Engine has run out of space for the GridCache, messages similar to the following message appear:

```
Fine: [GridCache] Pushing GridCacheRegion::trade_10985 out of the cache
```

To prevent this, you can increase the cache sizes. For Engines, the **Disk Cache Size**and **Memory Cache Size**settings are in the Engine Configuration. For Drivers, they are the `DSCacheMaxMemInMB` and `DSCacheMaxDiskInMB` properties in the `driver.properties` file.

Database Issues

GridServer has two databases: an internal database, only used internally by the Manager; and an optional external database for reporting. For information about installing the reporting database, see the *GridServer Installation Guide*. For a complete description of the reporting database schema, see Reporting Database Tables.

Connection Problems

If GridServer loses connection to a database, you see a message in the Manager log similar to this:

```
Warning: [DirectorReportingPlugin] Database is unavailable:
java.sql.SQLException: while creating new connection to :jdbc:oracle://
rcfdsp01.com:9158/, Connection refused
```

Use a tool like DBVisualizer to access the SQL database using the JDBC URL and Driver JAR to check if you can connect and access the database.

JDBC Problems

If GridServer cannot find the JDBC driver, a message similar to this appears:

```
error for postgres:

SQL block processing is ON:

        "begin" begins block

        "end" ends block

        "/" terminates block statement

        Overridden newline character separates statements in blocks

Performing Class.forName() on driver org.postgresql.Driver

Driver jar needs to be placed in the classpath

java.lang.ClassNotFoundException: org.postgresql.Driver

        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)

        at java.lang.ClassLoader.loadClass(ClassLoader.java:418)

        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:355)

        at java.lang.ClassLoader.loadClass(ClassLoader.java:351)

        at java.lang.Class.forName0(Native Method)

        at java.lang.Class.forName(Class.java:264)

        at
com.datasynapse.commons.sql.CreateDB.validateProperties(CreateDB.java:112)
```

```
        at com.datasynapse.commons.sql.CreateDB.<init>(CreateDB.java:52)

        at com.datasynapse.commons.sql.CreateDB.main(CreateDB.java:43)
```

The JDBC JAR needs to be in `DS_MANAGER/webapps/livecluster/WEB-INF/lib`.

Reporting Database Write Failures

If GridServer cannot write to the reporting database, it starts discarding events, and this message appears:

```
[BrokerReportingPlugin] Client Backlog full, event discarded
```

Also, the Manager might fail to commit a piece of reporting data, and this message appears:

```
Severe: [DirectorReportingPlugin] engine stat commit failed
```

This must not affect operation of the grid, as the reporting features are separate from the core functionality. If this message occurs once, it is most likely caused by a temporary connection problem with the reporting databas,e and can safely be ignored. If several messages like this occur, it indicates that the reporting database parameters need to be adjusted, especially if reporting data is relied on for statistical data.

Reducing the Transaction Isolation to `TRANSACTION_READ_UNCOMMITTED` improves the performance of inserts into the reporting database. Increasing the number of connections to the database does not help because GridServer has a set number of threads running to the database from the backlogs. Increasing the size of the backlogs helps deal with peaks of activity in the Manager. Increasing the commit interval to 30 or 60 seconds also improves performance. Use of auto commit is not recommended for a production grid.

Troubleshooting Tools

The following tools and utilities can be helpful when troubleshooting a GridServer issue.

Task Admin Page

You can view Service invocation status data on the **Task Admin**page, which can be helpful when troubleshooting Service execution. Normally, Service invocation data is purged after a Service has completed. This can be changed by setting the `PURGE_INVOCATION_DATA` Service option. This

data is stored in memory on the Broker, so this setting might need to be modified depending on how the Service is used. The following options are available:

- `INVOCATION_COMPLETED` — data for each request is purged as soon as the request has completed. In this case, the data is only available when the request is queued or being processed. This must be used when the Service might be a long-running open Service, as each entry requires memory.
- `INVOCATION_COMPLETED_SUCCESSFULLY` — data for each request is purged as soon as it has completed, but only if the request was successfully completed. This is useful when diagnosing failure of long-running Services.
- `SERVICE_COMPLETED` — data is purged when the Service instance has completed. As long as the instance is running, the data is available. This is the default.
- `SERVICE_REMOVED` — data is removed when the completed Service instance is removed from the Admin interface. This enables you to view data after an instance completes, at the cost of memory overhead on the Broker. The instance is removed either automatically according to the Service Cleanup settings on the Broker, or when it is manually removed.

Task Queue Dump

You can view a textual representation of the Task Queue, which can be helpful in debugging. The Task Queue Dump contains information about Service sessions and tasks including task counts, retries, and conditions.It can be accessed from the **Diagnostics > Manager Diagnostics**page, by selecting Task Queues, and requires a Broker on the Manager.

Enabling Enhanced Task Instrumentation

Normally, a submitted task or remote Service Invocation’s execution time is measured only from start to finish. But often it is useful to be able to track the time that is spent in the various stages of this process, including input serialization, disk writing, task message submission, task queuing, task fetching, data transport, input deserialization, task processing, output serialization, output transport, queuing, and so on. This helps you understand the timing characteristics of distributed computing, optimize the process, and diagnose problems with greater ease.

To enable enhanced task instrumentation

1. Go to **Admin****> System Admin > Manager**Configuration**>****Services**.
2. Under the **Instrumentation heading,**set **Enable**to `True`.
3. Click **Save**.

   When enabled, task instrumentation applies to all Services on the Manager.

Note:

Task instrumentation slows down the Manager, and also requires additional disk space, so it is important to disable it after you finish using it. It is NOT recommended for production systems.

To view data generated by enhanced task instrumentation

1. Go to **Services > Service Session Admin**.
2. Find the Service that you wish to view, and select **View Instrumentation**from the **Actions**menu. Note that this choice only appears after the Service finishes running.

A table appears showing data collected by enhanced task instrumentation for the Service. For more information about instrumentation, see the *GridServer Developer's Guide*.

Process Explorer

Process Explorer is a free Windows utility from sysinternals.com. It provides detailed information about a process. For a particular process, you can view the DLLs it has loaded and the resource handles it has open. This is useful when Engines stop responding, wait on files indefinitely, or DLL issues of applications running on Engines. A Find capability enables you to track down a process that has a resource opened, such as a file, directory or Registry key, or to view the list of processes that have a DLL loaded.

Dependency Walker

Dependency Walker is also very useful for troubleshooting system errors related to loading and executing modules. It detects many common application problems such as missing modules, invalid modules, import/export mismatches, circular dependency errors, mismatched machine types of modules, and module initialization failures. From Visual Studio or from `dependencywalker.com`, install `depends.exe`.

Event Streaming by Using Apache Kafka

Event streaming exposes the internal events that are generated in GridServer so that users can use the information for purposes such as monitoring. You can stream component-wise events by using Apache Kafka Message framework.

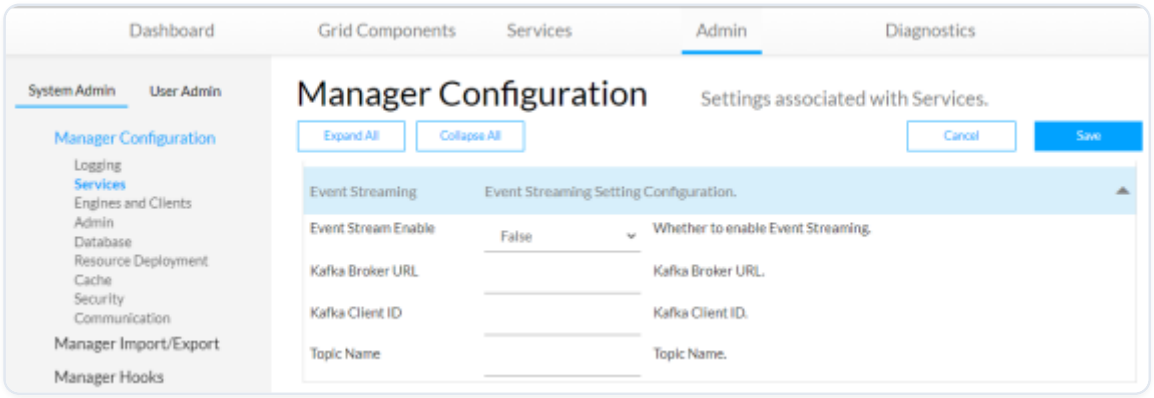
Configuring Event Streaming

Prerequisites

- You must have Apache Kafka running.
- GridServer Manager must be up and running.

Procedure

1. Log in to the GridServer Administration Tool with an account that has a Security Role with the Manager Configuration Edit feature enabled.
2. Go to **Admin > System Admin > Manager Configuration > Services > Event Streaming**.
3. On the Director, configure the values in the Event Streaming section. You can configure Kafka settings on the Primary and the Secondary Director. \



Manager Configuration

Refer the following table to configure the Manager:\

PROPERTY	VALUE
Event Stream Enable	Whether to enable event streaming. If it is set to `True`, you must also set the `Kafka Broker URL`, `Kafka Client ID`, and `Topic Name` properties.
Kafka Broker URL	Kafka Broker URL. The URL format must be: `Kafka installed Machine HostName:Port Number`.
Kafka Client ID	ID of the Kafka Client If you use event streaming, the Kafka Client ID value is appended with `_Broker` for a Broker and `_Director` for a Director. For example, if

PROPERTY	VALUE
	you enter the Kafka Client ID as `Grid_Client`, it is appended as follows: For Broker: `Grid_Client_Broker` For Director: `Grid_Client_Director`
Topic Name	Name of the topic

4. Click **Save**.

Events Captured by Apache Kafka

The following events are captured by using Apache Kafka for GridServer:

EVENTS	DESCRIPTION
Broker Added	Broker is added to the Director
Broker Removed	Broker is removed from the Director
Driver Added	Driver is added to the Broker
Driver Removed	Driver is removed from the Broker
Engine Added	Engine is added to the Broker
Engine Blacklisted	Engine is blacklisted
Engine Daemon Added	Engine Daemon is added to the Director
Engine Daemon Removed	Engine Daemon is removed from the Director
Engine Died	Engine dies (did not log off properly)
Engine Greylisted	Engine is greylisted
Engine Removed	Engine is removed from the Broker
Memory Warning	Server exceeds the configured threshold of memory usage
Service Session Cancelled	Service session is cancelled

EVENTS	DESCRIPTION
Service Session Completed	Service session is completed
Service Session Started	Service session starts
Task Assigned	Task is assigned to an Engine
Task Completed	Task is completed
Task Error	Task fails
Task Submitted	Task is submitted to the Broker

Note:

These events are recorded in the log file generated over Apache Kafka. You can also specify a custom log directory in the `server.properties` file of Apache Kafka. To capture streamed events, write your own consumer code.

Reporting Database Tables

GridServer uses a simple relational database to report grid processing events for historical analysis. This appendix describes the tables in the reporting database.

Data Type Mapping

The following table lists mappings for data types that vary depending on the database type.

DATA TYPE	MS SQL	ORACLE	POSTGRESQL	HSQL
bigint	bigint	int	bigint	bigint
timestamp	datetime	timestamp	timestamp	timestamp
text	text	clob	text	longvarchar
varchar	varchar	varchar2	varchar	varchar

DATA TYPE	MS SQL	ORACLE	POSTGRESQL	HSQL
identity	int not null identity	int (incl. sequence+trigger)	serial or bigserial not null	int generated by default as identity (start with 1) not null
bigidentity	bigint not null identity	int (incl. sequence+trigger)	bigint not null identity	int generated by default as identity (start with 1) not null

batches

Batches that have been scheduled or run.

COLUMN NAME	DATA TYPE	DESCRIPTION
server	varchar(255)	The Manager where the Batch resided or ran
batch_id	bigint not null	The Unique ID number of the Batch Entry
time_stamp	timestamp not null	Timestamp of the event
event	int not null	Event code
class	varchar(255)	Class in the Batch
execution_id	bigint	Unique ID number of the Batch Execution, if applicable
description	text	Description of the Batch Event

brokers

Information about all Brokers that have participated in this grid.

Primary key:pk_brokers(broker_id)

COLUMN NAME	DATA TYPE	DESCRIPTION
broker_id	int not null	Broker ID number
broker_url	varchar(255)	The Broker's configured base URL
broker_name	varchar(255)	The name of the Broker

broker_stats

All statistical reports from Brokers are stored in this table.

Primary key:pk_broker_stats(broker_id, time_stamp)

COLUMN NAME	DATA TYPE	DESCRIPTION
broker_id	int not null	The unique ID of the Broker
time_stamp	timestamp not null	Timestamp of the report
num_busy_engines	int	Number of Engines busy at report time
num_total_engines	int	Number of Engines logged in at report time
num_drivers	int	Number of Drivers logged in at report time
uptime_minutes	float	Time since Broker start in minutes
num_jobs_running	int	Number of Services running at report time
num_tasks_pending	int	Number of tasks pending (not yet assigned to Engines) at report time

driver_events

Reports from Brokers generated when a Driver logs in or out.

Primary key:pk_driver_events(driver_event_id))

COLUMN NAME	DATA TYPE	DESCRIPTION
driver_event_id	identity	ID of the Driver event
username	varchar(255)	Driver username
hostname	varchar(255)	The name of the host running the Driver
time_stamp	timestamp not null	Timestamp of the report
broker_id	int	The ID of the Broker where the event occurred
event	int	0 for an add, or the reason code for a remove – reason codes are in the event_codes table

engine_events

Reports from Brokers that are generated when an Engine is added or removed. For example, when an Engine logs in or logs out.

Primary key:pk_engine_events(engine_event_id))

COLUMN NAME	DATA TYPE	DESCRIPTION
engine_event_id	identity	The ID of the Engine event
engine_id	bigint not null	The unique ID of the Engine
instance_id	int	The number of the Engine instance
time_stamp	timestamp not null	Timestamp of the report
broker_id	int	The ID of the Broker where the event occurred

COLUMN NAME	DATA TYPE	DESCRIPTION
event	int	0 for an add, or the reason code for a remove – reason codes are in the event_codes table

engine_info

Administrative information for all Engines that have ever logged in to this Director.

Primary key:pk_engine_info(engine_id)

COLUMN NAME	DATA TYPE	DESCRIPTION
engine_id	bigint not null	The unique ID of the Engine
username	varchar(255)	The username used by the Engine
guid	varchar(255)	Another unique such as a MAC address
IP	varchar(255)	The IP address used by the Engine
install_date	timestamp	When the Engine was installed

engine_stats

Statistic reports from Engine Daemons.

Primary key:pk_engine_stats(engine_id, time_stamp)

COLUMN NAME	DATA TYPE	DESCRIPTION
engine_id	bigint not null	The unique ID of the Engine
time_stamp	timestamp not null	Timestamp of the report
cpu_utilization	float	%CPU total utilization

COLUMN NAME	DATA TYPE	DESCRIPTION
ds_cpu_utilization	float	%CPU used by DataSynapse processes
total_ram_kb	bigint	Installed RAM reported by the OS in kilobytes
free_ram_kb	bigint	Free RAM reported by the OS in kilobytes
disk_mb	bigint	Free disk reported by the OS in megabytes
num_invokes	int	The number of Engine processes currently running

event_codes

Mappings of event codes to descriptive text.

Primary key:pk_event_codes(code)

COLUMN NAME	DATA TYPE	DESCRIPTION
code	int not null	Number of the code
name	varchar(255)	Description of the code

jobs

Historical information about all Services that have been run by GridServer.

Primary key:pk_jobs(job_id, start_time)

COLUMN NAME	DATA TYPE	DESCRIPTION
	bigint not null	The Service ID
service_type_name	varchar(255)	The Service Type used for the Service

COLUMN NAME	DATA TYPE	DESCRIPTION
job_class	varchar(255)	The Service class being run. For example: Java: `examples.calculator.service.JavaCalc` Dynamic Library: `Calculator` .NET: `NETCalculatorService.dll:DataSynapse` Command: `ls`
start_time	timestamp not null	When the Service was started
end_time	timestamp	When the Service finished
job_status	int	Service status (see job_status_codes table)
num_tasks	int	Number of tasks in the Service
task_time_std	float	Standard deviation of task completion time, in seconds
task_time_avg	float	Mean task completion time, in seconds
priority	int	Service priority when submitted
end_priority	int	Service priority when complete
driver_username	varchar(255)	Submitting Driver user name
driver_hostname	varchar(255)	Submitting Driver hostname
job_name	varchar(255)	Optional descriptive Service name from Description
app_name	varchar(255)	Optional descriptive application name from Description

COLUMN NAME	DATA TYPE	DESCRIPTION
description	varchar(255)	Optional description from Description
dept_name	varchar(255)	Optional descriptive department name from Description
group_name	varchar(255)	Optional descriptive group name from Description
indiv_name	varchar(255)	Optional descriptive individual name from Description
broker_id	int	ID of Broker that ran the Service
task_time_min	bigint	Minimum task time, in milliseconds
task_time_max	bigint	Maximum task time, in milliseconds
session_size	bigint	Service Session size in kilobytes
input_size_total	bigint	Sum of task input size in kilobytes
output_size_total	bigint	Sum of task output size in kilobytes
gridlibrary	varchar(255)	The Grid Library used for this Service instance
gridlibrary_version	varchar(255)	The Grid Library version

job_status_codes

Mappings of event codes to descriptive text.

Primary key:pk_job_scodes(code)

COLUMN NAME	DATA TYPE	DESCRIPTION
code	int not null	Number of the code
name	varchar(255)	Description of the code

roles

Contains user roles.

Primary key:pk_roles(name)

COLUMN NAME	DATA TYPE	DESCRIPTION
name	varchar(255)	Name of the role
description	text	Description of the role
features	text	Features within the role
managers	text	Managers assigned to the role
ldapgroup	varchar(255)	LDAP group of the role
driver_info	text	Driver info for the role

tasks

Historical information about all tasks that have been run by GridServer.

Primary key:pk_tasks(task_rec_id)

COLUMN NAME	DATA TYPE	DESCRIPTION
task_rec_id	bigidentity	
job_id	bigint not null	Service ID
task_id	int not null	Task ID
engine_id	bigint	Engine that eventually ran the task

COLUMN NAME	DATA TYPE	DESCRIPTION
start_time	timestamp	When the task was started, or 1970-01-01 00:00:00.0 (UNIX epoch) if the task was never started
end_time	timestamp	When the task finished
task_status	int	Task status (see task_status_codes table).
num_reschedules	int	Number of times the task was retried
engine_instance	int	Number of Engine instance that ran the task
task_info	varchar(255)	Task information
broker_id	int	The Broker that handled the task
description	varchar(255)	Description of the task
input_size	bigint	Size of the Task's input in kilobytes
output_size	bigint	Size of the task's output in kilobytes
submit_time	timestamp	When the task was submitted

task_status_codes

Contains mapping of event codes to descriptive text.

Primary key:pk_task_scodes(code)

COLUMN NAME	DATA TYPE	DESCRIPTION
code	int not null	Number of the code

COLUMN NAME	DATA TYPE	DESCRIPTION
name	varchar(255)	Description of the code

user_events

Historical user events.

Primary key:pk_user_events(user_event_id)

COLUMN NAME	DATA TYPE	DESCRIPTION
user_event_id	identity	The ID of the event
server	varchar(255)	The Server where the event occurred
username	varchar(255)	The user recording the event
IP	varchar(255)	IP address of the user that caused the event
time_stamp	timestamp not null	When the event occurred
handler	varchar(255)	The Internal handler class that recorded the event
event	text	A description of the event

users

GridServer user accounts.

Primary key:pk_users(user_id)

COLUMN NAME	DATA TYPE	DESCRIPTION
user_id	identity	The ID of the user
username	varchar(255)	The user name
user_access	int	The account's user access

COLUMN NAME	DATA TYPE	DESCRIPTION
user_info	text	The account's user info
personalization	text	Stores information about table and UI personalization
roles	text	Roles assigned to the user

Scheduler Instrumentation Database Table

GridServer uses a simple relational database to report operations of GridServer Scheduler. The `scheduler_info` table is described here.

Scheduler Instrumentation Database Schema Data Type Mapping

The following table lists mappings for the data types that vary depending on the database type:

DATA TYPE	MS SQL	ORACLE	POSTGRESQL
varchar	varchar	varchar	varying
bigint	bigint	number	bigint

scheduler_info

COLUMN NAME	DATA TYPE	DESCRIPTION
type	varchar(20)	It has various types such as - Start-Episode, End-Episode, matchItems, Checkpoints, Waiting Job List, and Waiting Engine List
time	bigint	Timestamp
value	varchar(2048)	It contains value in JSON format

Engine Instrumentation Database Table

GridServer uses a simple relational database to report the Engine Balancing process. The `engine_ins` table is described here.

Engine Instrumentation Database Schema Data Type Mapping

The following table lists mappings for the data types that vary depending on the database type:

DATA TYPE	MS SQL	ORACLE	POSTGRESQL
varchar	varchar	varchar	varying
bigint	bigint	number	bigint

engine_ins

COLUMN NAME	DATA TYPE	DESCRIPTION
type	varchar(20)	It has various types such as - Engine Disallowed, Engine Selected, Engine to Broker, Engine Logoff, and Director Balancing
time	bigint	Time stamp
value	varchar(2048)	It has a different value stored as per the type

Developer Guide

- Typographical Conventions
- TIBCO GridServer® Application Development
  - TIBCO GridServer® Programming Options
    - Services
    - PDriver
    - Python
  - Resource Deployment
  - Logging and Debugging
    - Logging Overview
    - Viewing Engine Logs
    - Writing to Logs
    - Debugging Engines
    - Creating a Native Stack Trace in Linux
  - Notes for Java Developers
  - Notes for C++ Developers
    - Changing the C++ Compiler Used with CPPDriver
    - C++ Multithreading Requirement
    - Using Global Statics in C++ Service Code
    - Running both 32-bit and 64-bit Services on 64-bit Windows Daemons
    - Running Examples on Visual Studio
    - Other C++ Notes
  - Notes for .NET Developers
    - .NET Driver Upgrades
  - Notes for Python Developers

- Driver Installation
  - GridServer SDK Installation
    - The Java Driver (JDriver)
    - The C++ Driver (CPPDriver)
    - The Parametric Job Driver (PDriver)
    - The Python Driver (PyDriver)
    - The .NET Driver
    - The COM Driver
    - The R Driver
  - Driver Configuration
    - Configuring Multi-Interfaced Drivers
    - Driver Cleaner Configuration
    - Multiple Driver Instances
- Creating Services
  - Overview
  - Steps in Using a Service
  - Service Method Compliance
    - Java/.NET Services
    - C++ Services
    - Command Services
    - R Services
    - Python Services
  - Client Calling Conventions
    - Java/.NET Client
    - C++ Client
    - R Client
    - Python Client

    - Registering a Service Type
        - Container Binding
        - .NET AppDomains
        - .NET Framework Versions
    - Language Interoperability
        - Strings and Byte Arrays
        - Object Conversion from Strings and Byte Arrays
        - XML Serialization for Java, .NET, and R
        - Interoperable Types for XML Serialization
        - R Interoperability
        - Python Interoperability
    - Maintaining State
    - Initialization
    - Cancellation
    - Destruction
    - Service Instance Caching
    - Invocation Variables
- Accessing Services
    - Services
    - Proxy Generation
    - Service Options
    - Service Invocation Context
        - Setting Task Description
    - Shared Services
        - Creating a Shared Service
        - Limitations to Shared Services
        - Ending a Shared Service
        - Shared Services and Failover

- Broker Spanning Services
  - Enabling Broker Spanning on a Driver
  - Admin API Usage
  - Scheduling and Task Expiration
  - Administration
  - Broker Spanning Service Limitations
- Service Groups
- Data References
  - C++ Data References
  - Python Data References
- Service Collection
  - Collect After Submit
  - Deferred Collection (Collect Later)
  - No Collection (Collect Never)
- Engine Pinning
- Running a Driver from an Engine Service

- PDriver
  - Overview
  - Installing PDriver
    - Resource Deployment
  - PDriver Commands
    - The pdriver Command
    - The bsub Command
    - The bcoll Command
    - The bstatus Command
    - The bcancel Command
  - About PDS Scripts

        - PDS Basics
        - PDS Structure
        - The Depends Statement
        - The Include Statement
        - Lifecycle Blocks
        - The Options Block
        - The Discriminator Block
        - The Schedule Block
    - Variables, Types, and Expressions
        - Basics
        - Scoping
        - Variable Substitution
        - Expressions
        - Arrays
        - Built-in Variables
    - Statements
        - Built-in Commands
        - The If Statement
        - The For and Foreach Statement
        - Shell Directives in Heterogeneous Environments
    - Example
- Creating Grid Libraries
    - Overview
    - Grid Library Format
        - Variable Substitution
        - Versioning
        - Dependencies
        - Conflicts

    - Grid Library Loading
        - State Preservation
        - Task Reservation
        - Environment Variables and System Properties
    - Using Grid Libraries from a Service
    - Super Grid Libraries
    - C++ Bridges
    - JREs
    - R Grid Libraries
        - Building TERR Runtime Grid Libraries
    - Python Bridges
    - Python Grid Libraries
    - Windows Application Deployment
    - Grid Library Example
- GridCache
    - Overview
    - General Capabilities
        - API
        - Modes
        - Cache Configuration and Access
        - Data Storage
        - Attributes
        - Consistency/Synchronization
        - Cache Loaders
        - Cache Loader Write-through and Bulk Operations
        - Notification
        - Disk/Memory Caching
        - Cache Region Scope

        - Data Conversion Matrix
    - Using The GridCache API
    - Fault Tolerance and GridCache
- GridServer Design Guidelines
    - Data Movement
        - Principles of Data Movement
        - Data Movement Mechanisms
        - Data Movement Examples
    - Service or Task Duration
        - Engine Interruption and Smoothing
        - Summary
- The Admin API
    - Documentation for the GridServer Admin API
    - Using the Admin API over SOAP
    - Using Server Hooks
    - Using JMX
- Using Conditions
    - Conditions
    - Discriminator Conditions
        - Setting Discriminators in the Administration Tool
        - Setting Discriminators Programmatically
        - PDriver Discriminators
    - Affinity Conditions
        - Setting Affinity Conditions Programmatically
        - Setting Affinity Conditions in the Administration Tool
        - Task Affinity
    - Custom Discriminator and Affinity Conditions

- Dependency Conditions
  - Creating Dependencies
  - Administering Task Dependencies
- Queue Jump Conditions
- Descriptor Conditions
- EXTRAConditions
  - Using the EXTRACondition REST Interface
  - Setting EXTRAConditions
- Condition Sets
  - AND set
  - OR Set
  - Service Set
- Engine Properties
  - Intrinsic Engine Properties
  - Custom Engine Properties
  - Engine Session Properties
  - GPU Services Engine Properties
  - MIC Processor Engine Properties
  - NUMA Engine Properties and Configuration

- Extending GridServer
  - Manager Hooks
  - Engine Hooks
    - Engine Hook Example
    - Implementing Engine Hooks as a Grid Library

- Task Instrumentation
  - Overview
  - Syntax

        - Client
        - Action
        - Object
    - Phases
        - Driver-side
        - Engine-side
        - Broker-side
        - DDT file write
        - Native
    - Example Phases in a Service Execution
- The grid-library.dtd
    - The grid-library.dtd
- REST API Reference
    - BatchAdmin
        - batch-definition
        - batch-definition
        - all-batch-execution-info
        - all-batch-info
        - batch-count
        - batch-definition
        - batch-definition-names
        - batch-execution-count
        - batch-execution-ids
        - batch-execution-info
        - batch-execution-info-by-batch-id
        - batch-ids
        - batch-info
        - running-batch-execution-count

- scheduled-batch-count
- selected-batch-execution-info
- selected-batch-info
- available
- batch
- batch-execution
- finished-batch-executions
- finished-batches
- resume-batch
- schedule-batch-definitions
- suspend-all-batches
- suspend-batch

- BrokerAdmin
  - service-discriminator
  - service-discriminator
  - all-broker-info
  - broker-count
  - broker-info
  - engine-router
  - service-discriminator
  - service-discriminator-names
  - shared-brokers
  - available
  - driver-weight
  - engine-router
  - engine-weight
  - maximum-engines
  - min-idle-home-engines
  - minimum-engines

- shared-brokers
- DriverAdmin
  - all-driver-info
  - driver-count
  - driver-info
  - available
- EngineAdmin
  - all-engine-info
  - busy-engine-count
  - engine-count
  - engine-ids
  - engine-info
  - engine-info-by-properties
  - log-url-list
  - selected-engine-info
  - available
  - kill-all-engines
  - kill-engine
  - park-engines
  - unpark-engines
  - parked-engines
- EngineDaemonAdmin
  - all-engine-daemon-info
  - default-properties
  - engine-daemon-count
  - engine-daemon-ids
  - engine-daemon-info
  - engine-daemon-info-by-properties

- log-url-list
- selected-engine-daemon-info
- available
- default-property
- property
- property-by-properties
- restart-engine-daemon
- restart-engine-daemon-by-properties
- all-enabled
- all-start-mode
- configuration
- configuration-by-properties
- default-property
- directors
- directors-by-properties
- enabled
- enabled-by-properties
- instances
- instances-by-properties
- property
- property-by-properties
- start-mode
- start-mode-by-properties

- DriverManager
  - broker-url
- ManagerAdmin
  - broker-id
  - broker-name
  - broker-url

- build-version
- busy-engine-count
- category
- category-names
- director-id
- engine-configuration-names
- engine-count
- events
- finished-service-count
- license-info
- manager-value
- pending-invocation-count
- running-invocation-count
- running-service-count
- service-count
- subscriber-events
- subscribers
- value
- version
- available
- manager-value
- value
- subscribe
- unsubscribe

- ServiceAdmin
  - all-services
  - invocation
  - service
  - deploy-resources

- all-service-info
- blacklisted-engines
- completed-service-invocation-count
- finished-service-count
- invocation-count
- invocation-info
- pending-invocation-count
- pending-service-invocation-count
- registered-services
- running-invocation-count
- running-service-count
- running-service-invocation-count
- selected-invocation-info
- selected-service-info
- service-binding
- service-count
- service-ids
- service-info
- service-info-by-properties
- service-invocation-count
- task-expiration-event-count
- available
- list-resources
- register-service
- all-finished-services
- finished-service
- resource-exists
- expires
- priority
- unregister-service

- UserAdmin
  - user
  - role
  - user
  - all-roles
  - all-users
  - role
  - user
  - available
  - role
  - user
- Version
  - version-release-name
  - build-version

Typographical Conventions

The following table lists the typographical conventions used in this guide:

CONVENTION	USE
`TIBCO_HOME`	Many TIBCO products must be installed within the same home directory. This directory is referenced in the documentation as `TIBCO_HOME`. The default value of `TIBCO_HOME` depends on the operating system. For example, on Windows systems, the default value is `C:\tibco`.
`DS_INSTALL`	TIBCO GridServer® installs into a directory within `TIBCO_HOME` named datasynapse. This directory is referenced in the documentation as `DS_INSTALL`. The default value of `DS_INSTALL` depends on the operating system. For example, on Windows systems,

CONVENTION	USE
	the default installation directory is `C:\tibco\datasynapse`.
`DS_MANAGER`	The *Manager directory* contains the read-only software files; by default, it is a directory within `DS_INSTALL` named `manager`, and is referred to as `DS_MANAGER`. For example, on Windows systems, the default Manager directory is `C:\tibco\datasynapse\manager`.
`DS_DATA`	The *data directory* is the location of all volatile files used by the application server such as server properties and configuration. By default, it is a directory within `DS_INSTALL` named `manager-data`, and is referred to as `DS_DATA`. For example, on Windows systems, the default data directory is `C:\tibco\datasynapse\manager-data`.

TIBCO GridServer® Application Development

This section is your starting point for developing applications that use your GridServer installation. The document is divided into several sections to help you understand the principles of the GridServer system, and how to program applications utilizing GridServer.

TIBCO GridServer® Programming Options

There are several options available to you when you adapt your applications to use GridServer. The following sections describe how to use each of them.

Services

Services provide for remote execution of code in a way that is scalable, fault-tolerant, dynamic and language-independent. Services can be written in a variety of languages and do not need to be compiled or linked with DataSynapse code. There are client-side APIs to create Service Sessions using Java, C++, COM, R, and .NET. A Service object on a client can create and use a Service implemented in the same or another language. In the Service model, requests on the client are

routed over the network, ultimately resulting in invocations on a remote machine, and response values make the reverse trip.

With GridServer, Services are *virtualized*; rather than send a request directly to the remote machine hosting the Service Session, a client request is sent to the GridServer Manager, which enqueues it until an appropriate Engine is available. The Manager selects which Engine services a request. The first Engine to dequeue the request hosts the Service Session. Subsequent requests can be routed to the same Engine or can result in a second Engine running the Service concurrently. For information about how this decision is made see the *TIBCO GridServer® Administration*. If an Engine hosting a Service Session becomes unavailable, another takes its place. This mechanism, in which a single *virtual* Service Session is implemented by one or more *physical* Sessions (Engine processes) provides for fault tolerance and essentially unlimited scalability.

See Creating Services for details how to implement Services; Accessing Services explains how to use Services in your application.

PDriver

The Parametric Job Driver, or PDriver, is a Driver that can execute command-line programs as a parallel processing service using the GridServer environment. This enables you to write a simple script to run a program on several Engines, and return the results to a central location.

PDriver scripts, which are written in the PDS scripting language, enable you to run the same program on Engines several times with different parameters. A script is used to define how these parameters change.

One way PDriver scripts can achieve parallelism is to iteratively change the value of variables that are passed to successive tasks as parameters. A script can step through a range of numbers and use each value as a parameter for each task that is created. Or, a variable can be defined containing a list of parameters.

For more information see PDriver.

Python

The Python Driver, or PyDriver, is a Driver that can submit Python scripts for execution in the GridServer environment. This enables you to create Python scripts to run on multiple Engines and return the results to a central location. You can achieve parallelism by iteratively changing the script that is passed to successive tasks.

Resource Deployment

Service resource files that are used by Engines are centrally managed, starting at the Director. The centrally located resources on the Director are then synchronized to Managers, which then synchronize them with Engines.

Grid Libraries are the method of deploying resources to Engines. They are an archive containing a set of resources and properties necessary to run a Service, along with configuration information that describes how those resources are to be used. Grid Libraries can contain Java classes and JARs, native libraries, .NET assemblies, configuration files, Java system properties, Engine hooks, and alternate JREs needed to run a Service. They can also contain references to other Grid Libraries as dependencies. A Service Session can use a Grid Library by setting the appropriate options for the Service Type used by the session.

The `build` directory of the SDK includes an example ANT build script that can be used to build Grid Libraries. The examples in the SDK can be automatically packaged as Grid Libraries by using this script and included configuration files. Each Service example contains `grid-library.xml` and `build.properties` files. The `build` directory contains `build.xml`, `deploy.bat`, and `deploy.sh`, which parse the `grid-library-build-properties` files to create Grid Libraries.

For more information about packaging Grid Libraries, see Creating Grid Libraries. For information about deploying Grid Libraries, see the *TIBCO GridServer® Administration*.

Logging and Debugging

GridServer contains comprehensive logging facilities on Engines. This can be used to diagnose problems with Services running on Engines, and your application can write information to these logs. This section contains an overview of GridServer's log facility, plus information about using it from your application, and how to attach a debugger to an Engine if needed.

Logging Overview

GridServer uses the `java.util.logging` package for its internal logging, to provide diagnostic messages to the console and to file. This section covers how to access these logs, and how to interface with the loggers.

The logger uses the following log levels, in order:

LEVEL	DESCRIPTION
Severe	Indicates serious failures

LEVEL	DESCRIPTION
Warning	Indicates potential problems
Info	Displays informational messages
Config	Displays static configuration messages
Fine	Provides tracing information
Finer	Indicates a fairly detailed tracing message
Finest	Indicates a highly detailed tracing message

Typically, the Info level is sufficient for most purposes and is best to use on a busy grid. In some cases, you might need to log at Fine level to diagnose certain issues. For example, you must set the log level to at least Fine to log JMX statistics.

Finer or Finest levels must not be used unless you are debugging a detailed issue and you are directed to do so by TIBCO support, as they might degrade performance and introduce unnecessary logging that can make it more difficult for diagnosing problems. If you must use Finest or Finer on the Manager, consider only increasing the component level.

The log format is: `{timestamp} {level}: [{component}] {message}`

Only messages that are at or above the current log level are logged.

> **Note:**
>
> It’s considered unsafe to set all logging at a level below Info.

> An example of a log message:

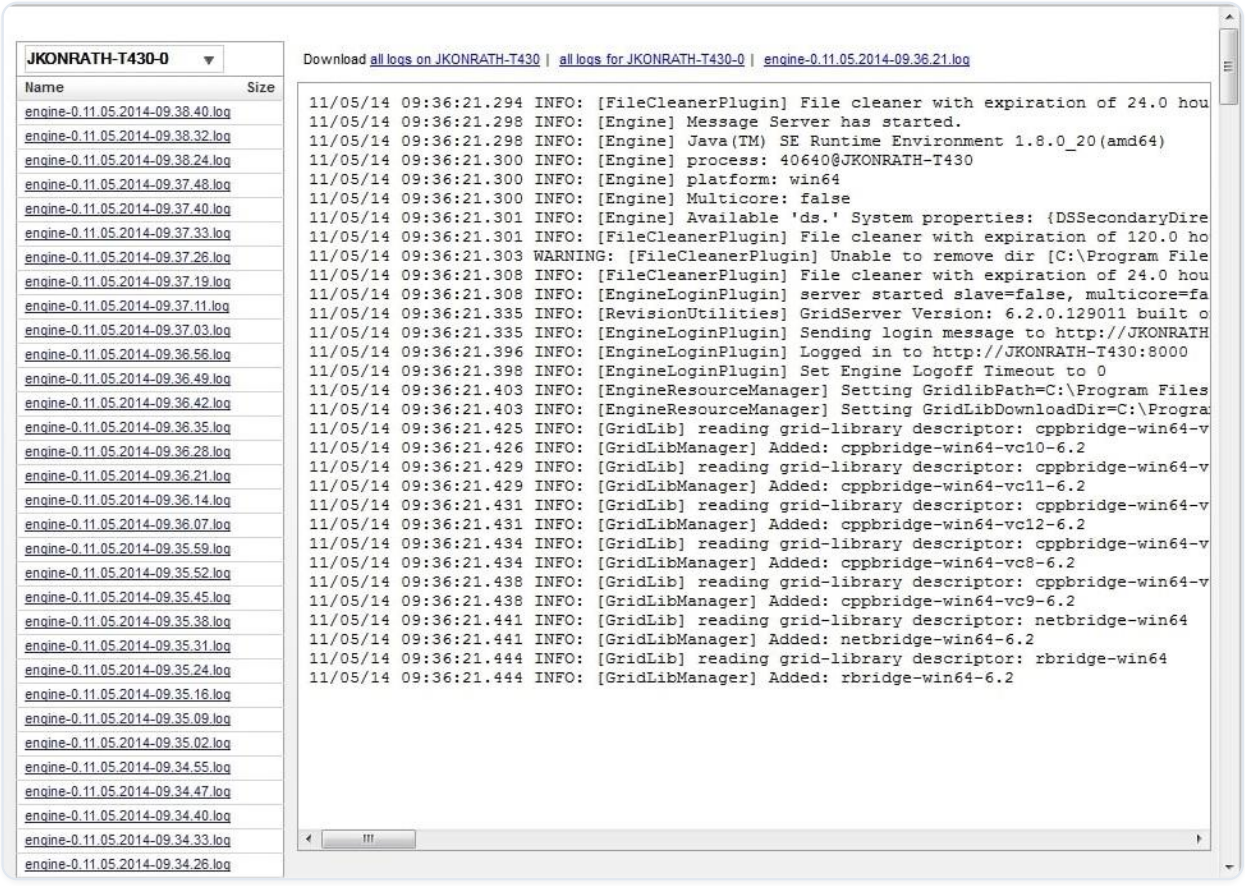
```
09/20/13 19:19:10.423 Info: [BrokerServicePlugin] Broker:Total:1
```

Viewing Engine Logs

There are several ways of viewing the logs. The most straightforward is to view the actual log files using the Log Files feature in the GridServer Administration Tool.

To view an Engine log:

1. In the Administration Tool, go to **Grid Components > Engines > Engine Admin**.
2. Select the Engine for which you want to view a log.
3. Click the **Actions**list, and select **Log Files**.
4. A window opens with a list of links for each of the logs residing on that Engine, listed by date and time. You can do any of the following:
   - Select an Engine Daemon or a particular Engine from the list in the upper left. This shows all of the log files on that Engine Daemon or Engine and their sizes. You can also type in the list box to quickly filter the list to partial matches.
   - Click on a log file name and its content is displayed to the right in the window.
   - Click the links in the upper right to download a ZIP archive of all log files on the host, a ZIP archive of all log files of an Engine Daemon or Engine instance, or a particular log file.



The Log Files window

You might also wish to view the logs in real time. You can do this with the remote log feature.

To view the remote log

1. Go to **Grid Components > Engines > Engine Admin**.

2. Select the Engine for which you want to view a log.
3. Click the **Actions**list, click **Remote Log**.
4. A window opens, displaying the log on the Engine as events occur. You can click **Clear**to clear the log, or **Snapshot**to capture a screen of the log in a new window.

You can also run the Engine in console mode; typically, this is only done during development.

Windows

You can run the Engine from a command line with the command `engine.exe -console.` This starts the Engine in console mode and logging information scrolls in the command window in which you start it.

UNIX

You can run the Engine from a command line with the command `engine.sh startfg`. This starts the Engine in the foreground and the logging information scrolls in the terminal in which you start it. Note that this is only suitable for debugging purposes.

The Engine Log Search page enables you to search for all Severe-level Engine logs for a Service ID across all Engines, and optionally search those results for a keyword. Results are shown with a summary of each matching log for each Engine, with links to corresponding URLs to logs with excerpts. First, logs are searched for the given Service ID; then they are searched for the regular expression "`.*Severe.*`", then they are optionally searched for a given keyword.

To search Engine logs

1. Go to **Diagnostics > Service Diagnostics**.
2. Enter a Service Session ID in the **Service Session ID**box, or click a name in the **Service Name**list. The list of Service Names is provided from the Service Admin list, which is the in-memory list of all Services recently run.
3. Enter a keyword in the **Keyword**box, or leave it blank to return all entries.
4. Click **Search**.

Results are shown with a summary of each matching log for each Engine, with links to corresponding URLs to logs.

Writing to Logs

Your Service also logs messages, and you can direct the messages to the DataSynapse logger.

Java

GridServer uses the Java logger, so any messages logged by using a Java Logger object are written to the Engine or Driver log file. For example:

```
Logger mylog = Logger.getLogger("com.mycompany.myproduct.MyClass");
```

The level of logs can be increased on a per-class and package level, to help in isolating issues. For example, you could do the following:

```
LogManager.getLogManager().getLogger("com.livecluster.admin.servlet.AdminC
ontrollerServlet").setLevel(Level.INFO);
```

Both Drivers and Engines capture `stdout` and `stderr`, so typically no changes are required to existing implementations to capture logs.

Additionally, the DataSynapse logger is registered as the Apache Commons Logging default handler. If your implementation uses this interface, your messages are logged automatically. The following is a map of levels to DataSynapse levels:

COMMONS	DATASYNAPSE
fatal	Severe
error	Severe
warn	Warning
info	Info
debug	Fine
trace	Finer

The Java Driver log manager can be completely disabled so that all Driver log messages are logged according to your configuration of the Java Logging Framework, rather than according to the Driver properties. For example, if the Driver is a part of a large client application that uses a number of Java libraries whose logs are all managed by the Java Logging Framework configuration, you would disable this so that the Driver logs are managed the same way. There is a Driver property, `DSLogUseJavaConfig`, that enables this behavior when set to `true`.

.NET

The .NET `System.Diagnostics.Trace` facility is used for logging; the DataSynapse logger is simply a Trace listener. The DataSynapse logger captures any messages written to the Trace facility. This includes .NET Services; any trace message written by the Service is logged to the Engine log.

C++

The `UtilFactory::log` function is the preferred method of logging to the DataSynapse log. You can use it on both the Engine and Driver.

> **Note:**
>
> For logging to become effective, it is necessary to instantiate the Driver message server first — for instance, by creating a Service object, or by calling `DriverManager::connect`.

R

The included `rlogger` package contains a `log` method for logging. For example, the following logging is used in the Pi tutorial included in the SDK:

```
library("rdriver")

library("rlogger")

...

RDPI <- function(N, K) {

rlogger.log(list("RDPI(", N, ", ", K, ") = ", pi), rlogger.FINE)
```

PDriver

The PDS script language provides redirection of `stdout` and `stderr` to a file, via the `stdout` and `stderr` clauses in the execute statement. For example:

```
    execute

        stdout="$DSWORKDIR/pijob.$DSTASKID.out"
```

```
        stderr="$DSWORKDIR/error.$DSTASKID"

        ".\resources\win64\lib\PdriverPiCalc.exe $seed $iterations"
```

Writing to the Log directory

The Engine's log directory is `[work directory]/log`. You can view files written to this directory with the Log Files feature. You can write log messages to your own files, and view them with the Administration Tool.

The work directory is available as follows:

- Java: The system property `ds.WorkDir`
- .NET: The `System.AppDomain.CurrentDomain` data value `ds.WorkDir`
- C++, Command Service: The environment variable `ds_WorkDir`
- PDriver: The variable `$DSWORKDIR`

C++/.NET Native stdout/stderr

Aside from the logging methods described above, it is possible to write native `stdout` and `stderr` to files that are managed by the Engine. Enable this with the **Logging for Native Streams**settings in the Engine Configuration. The log files are created in the Engine instance log directory, typically *install-dir* `/work/` *machine name* `-` *instance* `/log/*`. The file names are `engine-stdout-` *PID* `.log`, `engine-stderr-` *PID* `.log.` The files roll when they exceed the maximum size configured in the Engine Configuration.

This affects only native standard output / standard error for C++ and .NET. Java writes to `System.out/err` go to the regular invoke log file, as do writes from C++ or .NET that use the documented logging facilities.

Debugging Engines

This section covers the basics on how to attach a debugger to an Engine.

Java

The Java Platform Debugger Architecture (JPDA) enables the connection of a debugger to the Engine via a socket. You can set the socket used for debugging in the Engine Configuration. Create a new Engine Configuration or modify an existing one, and change the value of **Debug Start Port**. If you have a single Engine per Engine Daemon, set this value to the port you wish to use on each Engine. If you have more than one Engine per Engine Daemon, the value given in

Debug Start Portis the port used for the first Engine instance, and the port is incremented for each additional Engine Instance. You can also set **Debug Suspend**to true, which keeps the Engine suspended until a debugger is attached.

.NET, Windows DLL

Microsoft Visual Studio comes with a remote debugging facility. To debug, you must first make sure that you build with debug symbols, and deploy the symbols (PDB) file with the DLL. Once the Engine has logged in, you attach the debugger to the `invoke.exe` process via the **Processes**dialog on the **Debug**menu.

CPPDriver and Linux

GDB can be used to debug native code in CPPDriver or JNI in Linux. Also, GDB can be useful in identifying unusual problems with the Linux JVM. However, there are some subtle issues when trying to use GDB on a JVM, as is the case with the GridServer Engine.

When attaching GDB to the Engine, you must specify the `LD_LIBRARY_PATH` to both the Engine components and the JVM components.

```
LD_LIBRARY_PATH=lib:jre/lib/i386:jre/lib/i386/native_threads:jre/lib/i386/
server:resources/lib/linux
```

You must also obtain the process ID of a running `invoke` (or `invokeGCC34`) process from the `ps` command. It's also easier if you run GDB from the base directory of the Engine install (typically `DSEngine`). The GDB command used is similar to this:

```
gdb bin/invoke $INVOKEPID
```

Replace `bin/invoke` with `bin/invokeGCC34` when using GCC34.

One difficulty with debugging C++ code is that your application shared objects are loaded only when the Service is instantiated, so it becomes difficult to set a breakpoint in the application shared object. (However, more recent versions of GDB feature deferred symbol resolution, which makes this possible.) A technique that works in this instance is to have your application Service method include some conditional code to enter a loop checking some variable value that is never changed by the application code, effectively creating an infinite loop. When you need to attach GDB, trigger the conditional that causes the loop to be entered on the next invocation. Then attach GDB as above. You'll see that the invoke process is stopped while running in the loop. At that point, you

can change the loop evaluation value so that the infinite loop is exited, and the code continues to your breakpoint where you can continue debugging.

Creating a Native Stack Trace in Linux

Sometimes when you are troubleshooting native C/C++ code on Linux, you want to generate a stack trace, for example when a SIGSEGV is thrown. Since the JVM on the Engine already traps SIGSEGV and prints out a Java (not native) stack trace, you must override the actions of the JVM and install your own SIGSEGV handler for debugging. The `backtrace_fd()` and `backtrace_symbols_fd()` methods from glibc can be used for this purpose.

To install your own SIGSEGV handler for debugging, add code to your Service initialization method similar to this:

```
#include <execinfo.h>

#include <stdio.h>

#include <signal.h>

#define TRACE_DEPTH 50

void MyService::segv_handler(int signum) {

    void *trace[TRACE_DEPTH];

    int depth;

    FILE *fp;

    depth = backtrace(trace, TRACE_DEPTH);

    fp = fopen("trace.log", "w");

    backtrace_symbols_fd(trace, depth, fileno(fp));

    fclose(fp);

    abort();

}

void MyService::init() {
```

```
    signal(SIGSEGV, segv_handler);

    signal(SIGBUS, segv_handler);

}
```

Notes for Java Developers

Here are some notes for Java developers.

Stop Java Client by using a property

Unlike C++ and .NET clients, a Java client does not stop automatically. It remains pending after submitting Services and collecting results. Instead of calling `System.exit()` to stop the client, you can set a property to change this behavior. Add this line to your code:

```
DriverManager.setProperty(DriverManager.IS_DAEMON,
Boolean.TRUE.toString());
```

You can also enable this by setting `DSIsDaemon=true` in the `driver.properties` file.

When this property is set to true, all client threads are daemon threads, meaning that they allow the process to shut down when all threads have shut down.

HTTP Connection Parameters

The following parameters are specific for Java Drivers:

PARAMETER	DEFAULT VALUE	DESCRIPTION
`DSHttpMaxConnectionTotal`	500	It controls how many maximum concurrent connections the Driver application can have at any given time. To change the default value, you can specify `-DDSHttpMaxConnectionTotal=800` as JAVA OPTION (as one of the System properties) or specify the following

PARAMETER	DEFAULT VALUE	DESCRIPTION
		parameter: `DSHttpMaxConnectionTotal=`
`DSHttpMaxPerRoute`	80	It controls how many maximum concurrent connections per Route the Driver application can have at any given time. To change the default value, you can specify `-DDSHttpMaxPerRoute=100` as JAVA OPTION or specify the following parameter: `DSHttpMaxPerRoute=`
`DSHttpDDTKeepalive`	`True`	It controls whether to keep the HTTP connection alive for DDT requests. To change the default value, you can specify `-DDSHttpDDTKeepalive=false` as JAVA OPTION or specify the following parameter: `DSHttpDDTKeepalive=`
`DSHttpSubmissionKeepalive`	`True`	It controls whether to keep the HTTP connection alive for tasks submission requests. To change the default value, you can specify `-DDSHttpSubmissionKeepalive=false` as JAVA OPTION or specify the following parameter: `DSHttpSubmissionKeepalive=`
`DSHttpImplementation`	`com.livecluster.util.http.ApacheHttpClientSupportFactory`	It controls which HTTP communication implementation is applied. The default is `Apache HttpClient`. To change the default value to JRE HTTP Connection implementation, you can specify

PARAMETER	DEFAULT VALUE	DESCRIPTION
		`-Dds.DSHttpImplementation=com.livecl` as JAVA OPTION (System property) or specify the following parameter: `DSHttpImplementation=`

Notes for C++ Developers

Changing the C++ Compiler Used with CPPDriver

The CPPDriver and Service bridge libraries are built for nearly all standard compilers used on Windows and Linux. You must link your client application and/or Service implementation with the appropriate libraries for the compiler.

You must also run any C++ Services against the proper C++ bridge libraries. This is done using Grid Libraries, in that any C++ Grid Library must include the proper bridge Grid Library as a dependency These libraries come already deployed in the `DS_DATA/deploy/resources/gridlib` directory.

Also, because different Linux releases support different compilers which use incompatible versions of the STL, the **GCC Version**property in the Engine Configuration dictates which compiler version of the bridge is supported by the Engine.

To run Linux C++ Services against a specific compiler or STL version, set the **GCC Version**property in **Engine Configuration > Directories**to a supported version for your platform. The Manager's Product Info page lists the available GCC versions for each platform. The Engine automatically uses the corresponding C++ bridge libraries and invocation wrapper. You do not need to manually replace any files.

Notes for C++ Developers

Changing the C++ Compiler Used with CPPDriver

The CPPDriver and Service bridge libraries are built for nearly all standard compilers used on Windows and Linux. You must link your client application and/or Service implementation with the appropriate libraries for the compiler.

You must also run any C++ Services against the proper C++ bridge libraries. This is done using Grid Libraries, in that any C++ Grid Library must include the proper bridge Grid Library as a dependency These libraries come already deployed in the `DS_DATA/deploy/resources/gridlib` directory.

Also, because different Linux releases support different compilers which use incompatible versions of the STL, the **GCC Version**property in the Engine Configuration dictates which compiler version of the bridge is supported by the Engine.

To run Linux C++ Services against a specific compiler or STL version, set the **GCC Version**property in **Engine Configuration > Directories**to a supported version for your platform. The Manager's Product Info page lists the available GCC versions for each platform. The Engine automatically uses the corresponding C++ bridge libraries and invocation wrapper. You do not need to manually replace any files.

C++ Multithreading Requirement

Note that you must compile all UNIX C++ code multithreaded. This includes both Service code and Engine or Driver code.

Using Global Statics in C++ Service Code

By default, the Linux C++ Driver loads application libraries with `RTLD_LOCAL`, which only makes statics available within an object. This is done primarily to ensure that statics defined in a Service Session is unique.

However, this can cause issues in your application, such as when using dynamic cast with RTTI. Instead, you can use `RTLD_GLOBAL`, which makes statics available by any object in a process. There is an Engine Configuration option to set `RTLD_GLOBAL` to true to handle such cases.

Running both 32-bit and 64-bit Services on 64-bit Windows Daemons

The 64-bit Windows Engine Daemon can be configured to allow the execution of 32-bit Services. For more information, see "Configuring 64-bit Engine Daemons to run 32-bit Services" in the *TIBCO GridServer® Administration*.

Running Examples on Visual Studio

To run the Calculator client in windows using any Visual Studio version other than Visual Studio 2013:

1. Open the CPPCalculator solution file in Visual Studio.
2. Set the configuration to "Release" and Platform as "x64".

3. Right-click "Calculator", select **Properties > Configuration Properties > General > Platform Toolset**, and select your version of Visual Studio. Do the same for "CalculatorClient".
4. Right-click "Calculator", select **Properties > Linker > General**, and replace "Output File" with the value for your version of Visual Studio. (For example `..\..\target\o\win64\vc12\Calculator.dll`)
5. Right-click "CalculatorClient" select **Properties > Linker > General**, and replace "Additional Library Directories" with the value for your version of Visual Studio (For example `..\..\..\..\..\cppdriver\lib\vc12\x64;%(AdditionalLibraryDirectories)`)
6. Right-click "CalculatorClient", select **Properties > Linker > Input**, and replace "Additional Library Directories" with the value for your version of Visual Studio (For example `DSDriverVC12.lib;DSUtilVC12.lib;odbc32.lib;odbccp32.lib;%(AdditionalDependencies)`)
7. Modify `env.bat` for your version of Visual Studio. For example:

   ```
   set dsos=win64
   ```

   ```
   set dscompiler=vc12
   ```

   ```
   set dsbridgename=win64-vc12
   ```

   ```
   set devenvpath="%VS120COMNTOOLS%\..\IDE\devenv.exe"
   ```

8. Edit `driver.properties` as needed.
9. Run the `build-and-deploy.bat` script. The Grid Library is deployed with the correct `calculator.dll` file packaged in it.

Other C++ Notes

Microsoft Visual Studio 2013, 2015, 2017, and 2019 redistributables are installed when the Windows Engine is installed with the installer program. Engine updates from the Manager to pre-existing Engines do not install the runtimes.

When using existing Grid Libraries that use JNI on Windows systems, you must enable the `Restart on lib-path change` property in the Engine Configuration.

Notes for .NET Developers

The `GridServerNetClient.dll` references `GridServerNetBridge`, which isn't needed for clients. This causes a build warning about `GridServerNetBridge`, but can safely be ignored.

To throw a custom exception from a .NET Service and retain all data when the exception reaches the .NET client, the exception must implement `ISerializable`. For more information, consult MSDN.

Parts of the `BatchInfo`, `ServiceInfo`, `Options`, and `Discriminator` classes use variable / constant names that are not CLS compliant, and might not be usable from non-C# Framework languages. If this is an issue, contact DataSynapse technical support for a possible workaround.

.NET cache loaders for GridCache must be packaged as a Super Grid Library. For more information about using Super Grid Libraries, see the *TIBCO GridServer® Administration*.

For .NET5 users

For .NET5 users, .NET5 version 5.0.404 for Windows and Linux is supported. The example of .NET5 is under `examples\service\net5` directory and the client assemblies are in the `NET5Driver` directory. For the NET5 service, you must configure Service Type as NET5.

If you are running .NET5 service on Linux, you must change the `GCC build version` to `gcc49` in the Engine Configuration.

If your system does not support globalization and you get the following error:

`Couldn't find a valid ICU package installed on the system`,

add the following environment variable before running NET5:
`DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=true`

> **Note:**When you compile a .NET5 project, ensure that the outpath does not contain spaces. If the outpath contains spaces, then enter the outpath in double quotes (" "). For example,

```
-o "${basedir}\work\net5calculator-1.0.0.1\assemblies"
```

.NET Driver Upgrades

The .NET Driver (`GridServerNETClient.dll`) is strongly named. This means that when a new major version of the .NET Driver is released in an update, steps might need to be taken for existing clients to allow the assembly to be loaded. For example, that version changed from 5.1.x to 6.0; however, it has remained the same for all 6.x versions.

There are two ways of doing this:

- Rebuild the .NET application with the new `GridServerNETClient.dll`.

  or

- Configure the application to allow the new version.

You can do this in various ways, depending on your .NET policy; that is, whether the assembly is deployed into the GAC or used locally.

An example of how to do this when the `GridServerNETClient.dll` is used locally is as follows:

Method 1: Using the Microsoft .NET Framework Configuration tool:

1. Select **Start > Control Panel > Administrative Tools > Microsoft**.NET**Framework**x.y****Configuration**, where *x.y* is the version of .NET.

   Note that the tool is no longer included with Windows or in the .NET runtime; you must install the .NET Framework SDK to obtain the file.
2. Select **Applications > Add an Application To Configure**.
3. If your application is in the list, click it; otherwise, find it using the **Other…**button.
4. Your application is now in the Applications list. Expand your application, and select **Assembly Dependencies**.
5. Drag the **GridServerNETClient**, noting the version number, to the **Configured Assemblies**icon.
6. Click the **Configured Assemblies**icon. Double-click **GridServer****NETClient**, and select **Binding Policy**.
7. Under **Requested Version**, enter the version you noted in step 5. This is the version with which you built your application. Under **New Version**, enter the new version of the `GridServerNETClient.dll` that you just installed. This enables your application to bind with the new version even though you built it with a previous version.

Method 2: Directly creating the file

In Method 1, the .NET tool creates an Application Configuration file in the directory of the application. However, you can create this file yourself.

1. Create a file next to your application executable called `my.exe.config`, where `my.exe` is the name of your executable.
2. Add the following as the file's content:

```
<?xml version="1.0"?>

<configuration>

  <runtime>

    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

      <dependentAssembly>

        <assemblyIdentity name="GridServerNETClient"

                          publicKeyToken="42129437978483df" />

        <bindingRedirect oldVersion="5.0.0.1-5.1.2.30"

                         newVersion="6.0.0.1" />

      </dependentAssembly>

    </assemblyBinding>

  </runtime>

</configuration>
```

Note that the `oldVersion` is the version you built your application with, and the `newVersion` is the version of the new assembly. In this example, `oldVersion` is a range of versions. If your applications already have a configuration file, edit the configuration file appropriately.

If you have a .NET Service implementation that links to the `GridServerNETClient.dll`, you do not need to perform either of these steps. An `invoke.exe.config` file is included in any .NET upgrade that manages this for you. However, you can rebuild your implementation if you wish.

Notes for Python Developers

The Python client wraps the 64-bit VC14 version of the C++ Driver on Windows, so the 64-bit C++ Driver and Python (3.7.0) need to be in the path and the VC14 Runtime needs to be installed.

Driver Installation

This section describes how to install the GridServer SDK and Drivers and configure the Drivers.

The *Driver* is the component that maintains a connection between the GridServer Manager and the client application. The GridServer SDK, available for Windows and Linux, provides the Drivers.

GridServer SDK Installation

This section describes the GridServer SDK installation. To understand your complete installation procedure, read the SDK instructions and the instructions for each Driver you plan to install.

To install the GridServer SDK:

1. In the GridServer Administration Tool, go to the top navigation bar, and click the **Downloads**icon.
2. Click the SDK for your platform to download it.
3. Unzip or unarchive the SDK.
4. Read the following sections for directions on installing Drivers.

The Java Driver (JDriver)

The Java Driver, also known as JDriver, consists of a JAR file used with your Java application code.

To use JDriver:

1. Ensure that you installed the Java SE SDK (also commonly referred to as the JDK). You can download it from Oracle.
2. Define an environment variable `JAVA_HOME` that contains the location of the JDK.

Each of the Java examples in the `examples` directory of the GridServer SDK includes `env`, `build`, and `run` scripts. The examples demonstrate how to properly set classpaths and environment variables to run a Java application using JDriver.

To use the Driver, add the `DSDriver.jar` file and the `config` directory of the GridServer SDK to your classpath when running your application. Additionally, a `DSDriverNoDeps.jar` is included, which contains only DataSynapse classes and no third-party dependencies, allowing you to upgrade these dependencies if and when necessary.

The C++ Driver (CPPDriver)

The C++ Driver, also known as CPPDriver, consists of libraries and includes files that are linked with your GridServer application.

To use the C++ Driver:

1. Set the environment variable `DSDRIVER_DIR` to the path of the `config` directory in the SDK.
2. In Windows, set the `PATH` environment variable to your compiler `bin` directory.For example, for vc14, set the `PATH` environment variable to:`<install_dir>\GridServerSDK-Windows\cppdriver\bin\vc14`
3. In UNIX, set the `LD_LIBRARY_PATH` environment variable to your compiler `bin` directory.For example, for vc14, set the `LD_LIBRARY_PATH` environment variable to: `<install_dir>\GridServerSDK-Windows\cppdriver\bin\vc14`

The Parametric Job Driver (PDriver)

PDriver, or the Parametric Job Driver, is a Driver that can execute command-line programs as a parallel processing job using the GridServer environment. This enables you to take a single program, run it on several Engines, and return the results to a central location, without writing any new code.

The Python Driver (PyDriver)

To use PyDriver:

1. Add the 64-bit C++ Driver directory to the `PATH` environment variable.
2. Make certain the Python 3.7.0 directory is in the `PATH` environment variable.
3. For Windows, make certain the VC14 runtime is installed.

Windows Installation

To install PDriver on Windows systems, run the `PDriverInstaller.msi` installer in the `pdriver` directory of the SDK download. By default, this installs PDriver to the `c:\TIBCO\datasynapse\PDriver` directory.

This also installs the VC runtime if it is needed, and associates `.PDS` files with PDriver.

The installer sets the `DS_PDRIVER_CONF` environment variable, which specifies the location of the `driver.properties` file, to the installation directory. Verify that the `driver.properties` file contains the correct Manager and authentication settings so that the Driver points to your local Manager.

UNIX Installation

To install PDriver on UNIX systems

1. Copy the files in the SDK directory to your machine and set your path to include `pdriver/bin`.

2. Set the environment variable `DSDRIVER_DIR` to the path of the `config` directory in the SDK.
3. Set the `LD_LIBRARY_PATH` environment variable to include the `cppdriver/lib` and `pdriver/lib` directories.

> **Note:**
>
> Note that only Pdriver `linux64-gcc34` is supported from GridServer version 7.1.0 onwards.

Using PDriver

To run a PDriver script file, invoke the PDriver binary and pass the PDriver script file as the first argument. For example, on UNIX systems:

```
pdriver examples/example.pds
```

Running `pdriver` or `PDriver.exe` without a PDS argument provides a usage message.

The `examples` directory contains several example PDriver script files.

The .NET Driver

The .NET Driver consists of an assembly that includes classes for creating and managing Services from .NET. The .NET Driver is available only for Windows.

Perform the following steps to use the .NET Driver:

1. Find the `GridServerNETClient.dll` in the `NETDriver` directory and link this assembly to your application.
2. Set the environment variable `DSDRIVER_DIR` to the path of the `config` directory in the SDK.

.NET5 is also supported and is available for Windows and Linux. To use .NET5:

- Locate the `GridServerNET5Client.dll` file that is bundled in the SDK's `NET5Driver` directory and link this assembly to your application.

The COM Driver

The COM Driver enables an application using the Component Object Model architecture to access GridServer, enabling distributed parallel execution of the application on a Grid of Engines. The

Driver includes an example, an Option Evaluation spreadsheet in Excel that uses GridServer for its calculations. The COM Driver is available only for Windows.

Perform the following steps to install the COM Driver:

1. In the `COMDriver` directory of the SDK, double-click the `setup.exe` to start the installer.
2. After the installer welcome screen appears, click **Next**.
3. Click **Browse**to select a location for the COM Driver. The default location is `C:\TIBCO\DataSynapse\DSCOMDriver`. Click **Next**.
4. Enter the address of the primary and secondary Directors, in the form `hostname:port`. For example, http://server1.example.com:8000. You can proceed without filling in any values, but you must later specify your Directors by editing the `driver.properties` file. Click **Next**.
5. Click **Finish**.

The R Driver

The R Driver enables you to write Services in R, and access them from R or from applications in other languages such as Java, C++, or .NET. Applications written in R can also access Services deployed in GridServer.

Prerequisites

- Install and start the TIBCO GridServer® Manager.
- Install and start an Engine (Windows or UNIX).
- Download and extract the TIBCO GridServer® SDK for Windows or UNIX.
- Set the environment variable `DSDRIVER_DIR` to the path of the config directory in the SDK.
- For Windows:

  ```
  set DSDRIVER_DIR=GRIDSERVERSDK-Windows\config
  ```

- For UNIX:

  ```
  export DSDRIVER_DIR=GridServerSDK-Linux/config/
  ```

- Install TERR 6.0.
- Set `TERR_HOME` and `R_HOME` to the `TERR` install folder.
- For Windows:

  ```
  set TERR_HOME=C:\TIBCO\terr60
  ```

`set R_HOME=C:\TIBCO\terr60`

- For UNIX:

  `export TERR_HOME=/opt/qa/TIBCO/terr60`

  `export R_HOME=/opt/qa/TIBCO/terr60`

- Install Ant 1.9 or later and set `ANT_HOME` to the Ant install folder.
- Add `%TERR_HOME%\bin` and `%ANT_HOME%\bin` to the system path.
- For Windows:

  `set PATH=%PATH%;%TERR_HOME%\bin`

  `set PATH=%PATH%;%ANT_HOME%\bin`

- For UNIX:

  `export PATH=$TERR_HOME/bin:$PATH`

  `export PATH=$ANT_HOME/bin:$PATH`

- For Windows, add the directory `cppdriver/bin/vc15` to the `PATH` environment variable:

  `set PATH=%PATH%;GridServerSDK-Windows\cppdriver\bin\vc15\x64`

- For UNIX, set the `LD_LIBRARY_PATH` environment variable to include the applicable directory in the lib directory.

  For example, to use GCC3.4 libraries, set the variable to `cppdriver/lib/gcc34`:

  `export LD_LIBRARY_PATH=GridServerSDK-Linux/cppdriver/lib/linux64/gcc34/`

Installation

1. Run the following command to build and upload Grid libraries:

   - For Windows, go to `GridServerSDK-Windows/tools/bin` and run `createTERRGL.bat`:

`createTERRGL.bat --terrHome=` *%TERR HOME%* `--64 --complete`

   - For UNIX, go to `GridServerSDK-Linux/tools/bin` and run `./createTERRGL.sh`:

`./createTERRGL.sh --terrHome=$TERR_HOME --64 --complete`

1. Install `rdriver` and `rlogger`:

- For Windows:

```
TERR.exe CMD INSTALL ".\GridServerSDK-Windows\rdriver\rdriver"
```

```
TERR.exe CMD INSTALL ".\GridServerSDK-Windows\rdriver\rlogger"
```

- For UNIX:

```
$TERR_HOME/bin/TERR CMD INSTALL $SDK/rdriver/rdriver
```

```
$TERR_HOME/bin/TERR CMD INSTALL $SDK/rdriver/rlogger
```

2. Change directories to
   - For Windows:

     `GridServerSDK-Windows/examples/service/r/pscl`
   - For UNIX:

     `GridServerSDK-Linux/examples/service/r/pscl`
3. Modify the `build.xml` to change `/bin/Rscript` to `/bin/TERRscript`.
4. Edit the `build.properties` in the `pscl` folder. Set the `dsos` property to `Win64`, save the file. Run `ant`. This builds the Win64 version of the `Rextras` and `Rpscl` Grid Libraries.
5. Repeat the step 5 and set the `dsos` value to `linux64`. This builds the Linux64 version of the `Rextras` and `Rpscl` Grid Libraries.
6. Upload the four Grid Libraries, created earlier, (two `Rextras` and two `Rpscl`) to the Manager and deploy them. Deploy the Rextras first, then Rpscl.
7. Change directories to
   - For Windows:

     `GridServerSDK-Windows/examples/service/r/picalculator`
   - For UNIX:

     `GridServerSDK-Linux/examples/service/r/picalculator`
8. Run `ant` to build and deploy the `Rcalculator` Grid Library.
9. Run `runTERR.bat`. (This is only for Windows.)
10. Edit Service Type registry `RPICalculatorExample` on the UI and select `functionInterface` as `UNWRAP_RETURN`.
11. For Windows, run `runJava.bat` and for UNIX, run `runJava.sh`.

Driver Configuration

Driver configurations reside in the `driver.properties` file, which is included in the GridServer SDK. When you install the Manager, the `driver.properties` file contains hostnames for the primary and secondary Director. To change the default values, edit the `driver.properties` file. For example, edit this file to add a username and password for authentication, change the primary and secondary Director, change the Broker Timeout, and so on.

You can move the `driver.properties` file to another directory. For Java, add the new directory to your classpath. For C++, .NET, R, and the UNIX PDriver, the `DSDRIVER_DIR` environment variable must be set to the location of this directory. For Windows PDriver, the `DS_PDRIVER_CONF` environment variable must be set to the location of this directory.

If you use the following characters in properties in the `driver.properties` file, precede each with a backslash: `#`, `!`, `#`, `=`, `\`, and `:`

Backslash characters in hostname or directory properties receive special handling on Windows Drivers. The first backslash, indicating the root directory, translates as the current Windows drive. Other backslashes are ignored. Forward slashes translate as backslashes.

For example, to set a directory to `c:\sdk\log`, use `/sdk/log` in the `driver.properties` file. To use a UNC path such as `\\homer\job1-dir`, use `//homer/job1-dir` in the `driver.properties` file.

You can also set Driver properties programmatically with the API. When you do this, the `driver.properties` file is unnecessary. You must at least set `DSPrimaryDirector`, `DSSecondaryDirector` (set them both to the same hostname when using a single Director), `DSUsername`, and `DSPassword` (unless using Windows or Kerberos authentication).

If your username or password is non-ASCII, you must express it in Unicode converted format rather than as the Unicode characters. For example:

```
DSUsername=\u6B21\u306E\u30E6\u30FC
```

Configuring Multi-Interfaced Drivers

In some network configurations, on a PC with more than one network interface, a Driver might default to using the incorrect interface, resulting in the Engines using the incorrect IP address for the Driver's file server. To configure the Driver to use a different network interface, set the `DSLocalIPAddress` property to the IP number of the correct interface. For example:

```
DSLocalIPAddress=192.168.12.1
```

Driver Cleaner Configuration

When Direct Data Transfer (DDT) is enabled and data is transferred directly to the Engines from this Driver, DDT files are persisted for a configurable amount of time, and then deleted by a file cleaner process. The file cleaner deletes files and directories when they are no longer needed, such as files left over by a Service that did not complete normally. Files used by a currently executing Service are not deleted regardless of the files' age or the Driver's configuration.

There are four properties in the `driver.properties` that control the cleaner:

- `DSDataTransferFileExpirationHours`: The amount of time DDT files persist before being deleted. By default, this is set to 120 hours.
- `DSDataTransferFileExpirationFrequency`: The number of times per day that the Driver checks for and cleans expired DDT files. By default, this is set to 6 times. To disable the file cleaner, set this to 0.
- `DSDataReferenceFileExpirationHours`: The amount of time Data Reference files persist before being deleted. By default, this is set to 120 hours.
- `DSDataReferenceFileExpirationFrequency`: The number of times per day that the Driver checks for and cleans expired Data Reference files. By default, this is set to 24 times. To disable the file cleaner, set this to 0.

Multiple Driver Instances

Services can fail if you enable Direct Data Transfer and write a script that instantiates multiple Drivers from the same `driver.properties` file with the same port number. The first Driver opens a web server listening to the defined socket. Subsequent Drivers do not open another web server as long as the first Service is running, but can continue running by using the first Service's server for direct data. However, when the first Service completes, its server ends, causing subsequent Services to fail.

You can avoid this problem by writing a shell script to create Services, each with its own Driver running from its own Java VM. Your script must provide a different port number for the `DSWebserverPort` property normally set in the `driver.properties` file. To write a shell script for this situation, you can remove the `DSWebserverPort` property from the `driver.properties` file and assign a unique port number for each iteration.

More than one Driver can share the same directory. However, if you do so you must set a unique `DSCacheRootDirectory` for each Driver if using GridCache. Also, if using long-running Services, you must set a unique `DSWebServerDir` property so that one Driver's file cleaner does not clean another session's initialization data.

Creating Services

This section provides information about how to create Services.

Overview

Services enable remote, parallel execution of code in a way that is scalable, fault-tolerant, dynamic, and language-independent. You can write Services in a variety of languages and do not need to compile or link them with DataSynapse libraries. There are client-side APIs in Java, C++, COM, R, Python, and .NET. A client written in one language can invoke a Service written in another.

The Service execution model is the same as that of other distributed programming solutions: method calls on the client are routed over the network, resulting in method calls on a remote machine, and return values make the reverse trip. We prefer the term *request* to call or invocation, partly because the operation can be either synchronous or asynchronous.

Use Services to implement parallel processing solutions in which a single computation is split into multiple, independent pieces whose results are combined. To split the computation, divide the problem, submit the individual requests asynchronously, then combine the results as they arrive. Services also work well for executing multiple, unrelated serial computations in parallel.

Steps in Using a Service

Using a Service involves the following steps:

1. **Write the Service, or adapt existing implementations.**A Service can be virtually any type of implementation: a library (DLL or .so), a .NET assembly, a Java class, an R function, a command, script or executable, or even an Excel spreadsheet. You do not need to link a Service with any DataSynapse libraries, but make sure that the remotely callable methods of the Service follow conventions that enable cross-language execution and support stateful Services. For details, see Client Calling Conventions.

   For examples of code using Services, see the GridServer SDK and the *GridServer Service-Oriented Integration Tutorial*.

2. **Deploy the Service.**Make the implementation and other resources required for the Service accessible from all Engines. Do this through GridServer's mechanism of resource deployment.
3. **Register the Service Type.**Make the Service visible to clients by registering it as a Service Type in the GridServer Administration Tool.
4. **Create a Service Session from a Client.**Develop a Client Application that accesses the registered Service Type and creates a Service Session. Each Service Session has its own state that is client-specific.
5. **Make requests.**The Client Application calls the methods of a Service Implementation either synchronously or asynchronously.
6. **Destroying the Service Session.**Client Applications destroy Service Sessions when they are done with them.

This section describes how to develop a Service Implementation that runs on an Engine. Accessing Services describes how to use this Service Implementation from a Client Application.

Service Method Compliance

Although Service methods do not link to DataSynapse libraries, they must comply with the rules in the following sections.

Java/.NET Services

Java/.NET Services must comply with these rules:

- Make the Service class public, and make all methods called by the client public.
- A method can take any number of arguments and can have a return type of void. The return values of state methods are ignored, and a Service method with a void return type returns a null.
- If you plan to use the Service cross-language, the arguments and return values must conform to the rules of interoperability, as described in the section Interoperable Types for XML Serialization.
- If only a client of the same language uses the Service, you can use any serializable object for arguments and return values.
- Overloaded methods are prohibited.
- Methods can throw exceptions within a Service, which captures and includes stack trace data and nested exception data when available.

C++ Services

C++ Services must comply with these rules:

- Export all service methods in the shared library.
- The Service method must either take a `char*`, or a `char**` for multiple arguments. Alternatively, if using the macro it can take a `std::string` or a vector of `std::string` s.
- The method returns data through a `char**` argument, which is set to the returned data. Alternatively, if using the macro it returns a `std::string`.
- Overloaded methods cannot be used.

Command Services

Command Services must comply with these rules:

- The name of the method that is called is appended to the command line.
- Argument values are sent to stdin or an input file. If there is more than one argument, the data is separated by the `argDelimiter`, which can be registered on the Service.
- You can append argument values to the command line instead, if the option is selected. In this case, the client passes the argument values in as strings.
- If your command spawns subprocesses, it must cancel them if the main command is canceled.

R Services

R Services must comply with these rules:

- All functions visible in the R runtime environment can be used as a Service.
- These include all functions defined in any `.R` script files defined in the `R/` directory in the Grid Library and any function exported by third-party libraries.
- Functions exported by libraries can only be accessed when using `functionInterface` as `NATIVE_R`. The `functionInterface` setting is described in R Interoperability.

Python Services

Your Python libraries not only need to be added to the `grid-library.xml` for the Python bridge but also the PIP install needs to be done in your code. Your Python libraries need to use the Python Service Type to ensure that the proper Python script is executed.

You must carefully handle type conversion since the Python driver wraps the C++ Driver and the data is being implicitly converted to a string before being transmitted, and then it is implicitly converted from the string in the Grid Library unless explicit conversion is used.

Client Calling Conventions

Clients must comply with the following rules when calling methods in a Service.

Java/.NET Client

Java/.NET Clients must comply with the following rules when calling methods in a Service:

- Pass arguments into calls as `Object[]`, which corresponds to the arguments of the method. Note that you must use exactly the type `Object[]`. For instance, a set of strings cannot be passed in as a `String[]`. The array length must match the number of arguments.
- For convenience, if the method takes only a single argument, you pass it directly into the call. It is the equivalent of passing in an `Object[1]` with the 0th element being the object.
- If a method takes no arguments, you can call it only with zero-length `Object[]` or a null object.
- Autoboxing is used for primitive types. For example, a Service method that returns a `Double` on the client.

C++ Client

C++ Clients must comply with the following rules when calling methods in a Service:

- Pass arguments into calls as `char**`. Alternatively, if using the macro found in `DynamicLibraryFunctions.h`, it can be a vector of `std::string`.
- If a method takes no arguments, you can call it only with a `NULL` or zero-length `char*` or `string.`

R Client

See R Interoperability for more information.

Python Client

When setting parameters with the Python client, you must initialize the input parameters using `driver.init_input_parameter(` *n* `)` where *n* is the number of parameters to initialize. Each parameter is then set with a call to `driver.set_input_parameter(` *parameter name or literal* `)`. Here is an example from `python_client_example.py`:

```
driver.create_services(b'PythonExample')

driver.init_input_parameter(2)

driver.set_input_parameter(b"8.23")

driver.set_input_parameter(b"1.25")

driver.submit(b"add")
```

In the above example, the literal values are expressed as `'b"8.23"'` and `'b"1.25"'`. These are expressions specific to Python. Refer to Python documentation if you have questions on them.

Registering a Service Type

Service Types are registered in the GridServer Administration Tool. On the primary Director, go to **Services > Services > Service Types**. Service Types registered on the Director are then replicated to Brokers. A list of existing Service Types appears on that page, along with a line for adding a new Service Type. Enter the Service Type name on the blank line. Select a Service Implementation, then click **Add**.

In the window that appears after clicking the **Add**button, enter any name, property or option values for the Service Type.

Container Binding

When you register a Service Type, an associated Container Binding is created. The Container Binding binds the Service implementation (the library or command) to the Container of the Service (the Engine). The Container Binding describes how to use the implementation.

The binding contains the following fields:

Container Binding Fields

FIELD	DESCRIPTION
`initMethod`	Called when a Service Session is first used on an Engine. It is called prior to any requests or updates.
`destroyMethod`	Called when a Service Session is destroyed.

FIELD	DESCRIPTION
`cancelMethod`	Called when the Service is canceled if `KILL_CANCELLED_TASKS` is false for this Service. Use this field to interrupt the request if the user does not want the Engine to restart on a cancel.
`serviceMethods`	Methods that perform a request and return a response. These are the actual methods that perform the calculations. Use the * character to denote all methods that are not bound to any other action.
`appendStateMethods`	Updates state, appending to previous updates.
`setStateMethods`	Updates state, and flushes the list of previous updates.

You must bind all methods used to one of these methods. All methods are optional except `serviceMethods`.

The binding also contains the following fields, only applicable to Java, .NET, and R:

Additional Container Binding Fields For Java, .NET, and R

FIELD	DESCRIPTION
`xmlSerialization`	Whether to use XML serialization to serialize objects. See Interoperable Types for XML Serialization for more details.
`TargetPackage`	The package (Java) or namespace (.NET) into which the generated proxy classes are placed. If not set, the default is the name of the Service. This is only necessary when using generated proxies. In this case, if not set, serialization errors might occur. Note: When you do not set the `targetPackage` name in the **Service Types**page and use a generated proxy, deserialization errors occur. To remedy this, edit the **Service Type**on the **Service Types**page of the GridServer Administration

FIELD	DESCRIPTION
	Tool and assign a value for the `targetPackage` property.

The binding for R contains the `functionInterface` field, which is described in R Interoperability.

Python Services need to use the `Python` Service type. The only other critical step is making certain to set the `pythonModule` which is the name of the Python file that contains the implementation, without the file extension.

.NET AppDomains

Services implemented in .NET have full access to .NET's AppDomain functionality. You can manage multiple persistent AppDomains across Service invocations while still having access to the entire DataSynapse Engine-side API. You can specify an AppDomain as part of a Grid Library deployment, and the Engine sets up and manages it automatically.

The **Provider**section in the Service Type Registry for .NET Services supports an `appDomainName` value. Set the `appDomainName` value to specify a unique AppDomain for Services created from this Service Type. If you do not specify an AppDomain for a Service, the Service receives a new anonymous AppDomain with a randomized name. The Service does not run in the Default Domain as it did in previous versions of GridServer. If you create a Service that uses legacy Default Resources rather than a Grid Library, the Service runs in an auxiliary AppDomain. All resources-backed Services use this same auxiliary AppDomain for the lifetime of the Engine.

When you specify an AppDomain as part of a Service Type definition and an Engine creates the Service for the first time, the Assembly search paths used for the AppDomain depend on how you deploy resources:

- When a Service uses a Grid Library, the `assembly-path` path elements for that Grid Library (and any dependent Grid Libraries) are the Assembly search paths for the AppDomain.
- When a Service does not use a Grid Library, the default Assembly search path is used.

The Engine searches the Assembly search path for a valid AppDomain Configuration File, which has the same name as the AppDomain, plus the `.config` suffix.

An Engine restarts if it needs to load a Grid Library to run a Service in an AppDomain that has an Assembly search path that conflicts with the new Grid Library's search path. This is different behavior from previous versions of GridServer.

Use the `unloadAppDomain` property in the Service Type to specify what to do with Grid Library-backed AppDomains after the Service Sessions using them are destroyed. Select `true` to unload AppDomains no longer in use.

.NET Framework Versions

In the **Provider**section of the Service Type Registry is a `frameworkVersion` value. Set this value to the .NET Framework version required by Service. This prevents Services that specify a Framework version from executing on Engines without that framework version installed.

To determine which Engines have a version of the Framework installed, go to **Grid Components > Engines > Engine Admin**, select the **Engine Details**action, and look for the `NETFrameWorkVersion` and `NETFramework` Engine Properties. The `NETFramework` property is `false` if a Framework is not installed or if it is an unsupported version (such as 1.0).

.NET5 Core

For .NET5 users, select NET5 from the **Service Type**drop-down list.

Language Interoperability

Services provide various levels of interoperability among languages. To provide this interoperability, GridServer can perform conversions on arguments sent to objects. The following sections describe the conversion of arguments between Service Implementations.

Strings and Byte Arrays

All Services can use byte arrays (`byte[]` s) interchangeably with `String` s as arguments. Conversions use UTF-8 encoding. For example, if an argument is of type `String`, and the client passes in a `byte[]`, the `byte[]` is UTF-8 encoded and passed into the method as a `String`.

Because a C++ Service always returns a `string/char*`, you must convert the returned type of an invocation to a `String` or `byte[]`. The first argument to the invocation determines the type of conversion:

- A string first argument returns a string.
- An argument of `byte[]` returns a `byte[]`.
- If you pass no arguments to the invocation, it returns a `byte[].`

This is most relevant for a .NET, as string I/O must be ASCII. If you are returning binary data, make sure that the first argument is a `byte[]`.

A Command Service converts the output data to a `String` only if the first argument is a `String` and the `appendArgsToCommandline` option is false.

Java and .NET Services do not convert return values if they are `String` s or `byte[]` s.

In the case where a `parameterType` value is specified, GridServer converts to the specified value. For example, if `parameterType` is set as `string`, GridServer returns strings. If set to `byte[]`, GridServer converts the return value to `byte[]`. This applies to both the C++ Service and the Command Service. You can specify the `parameterType` value for DynamicLibrary and Command Services on the Service Type Registry page.

Object Conversion from Strings and Byte Arrays

Java and .NET Services automatically attempt to convert `String/byte[]` s to and from `Object` s when necessary. This is useful when calling these Services from a different language, or when using Service Runners from Batches.

If an argument is not a `String` or `byte[]`, and it is passed in as such, an attempt is made to convert it. If the data is a `byte[]`, it is first converted to a `String`. Then the `String` is converted to the `Object` as follows:

String Argument to Object Conversion

INPUT ARGUMENT	STRING ARGUMENT-TO-OBJECT CONVERSION
Primitives	The primitive wrapper class's parse method
`Date`, `Calendar` (Java)	`DateFormat.getDateTimeInstance().parse`
`DateTime` (.NET)	`DateTime.Parse`
`org.w3c.dom.Document` (Java)	Uses the parse method from the DocumentBuilder given by `DocumentBuilderFactory.newInstance().newDocumentBu`
`XmlDocument` (.NET)	`XmlDocument.loadXml`
Other	If the class has a constructor that takes a single `String` as an argument, it uses that constructor.

If the return value is not a `String/byte[]`, and the client is not of the same language as the Service, the returned value is converted to a `String`, as follows:

Returned Object to String Conversion

RETURN TYPE	RETURNED OBJECT-TO-STRING CONVERSION
Primitives	The object-equivalent `toString()` method
`Date`, `Calendar` (Java)	`DateFormat.getDateTimeInstance().format`
`DateTime` (.NET)	`date.ToUniversalTime().ToString("r", DateTimeFormatInfo.InvariantInfo)`
`org.w3c.dom.Document` (Java)	The transform method from the Transformer given by `TransformerFactory.newInstance().newTransformer()`
`XmlDocument` (.NET)	`doc.WriteTo(XmlTextWriter)`
Other	The `toString` method

XML Serialization for Java, .NET, and R

XML serialization provides the following features:

- Java, .NET, and R can use rich objects as arguments and return values with each other.
- A client can use a Service with such objects, without needing the original implementation classes. This is because client-side proxy classes are generated.

To use XML serialization, enable it on the Service Type. Note that when you enable this, the other interoperability conversions are no longer used. Additionally, the client must use the proxy that is generated using the Service Type Registry, which contains all user-defined types.

You must use Interoperable Types for the arguments and return values on such Services, as discussed in the following section.

Interoperable Types for XML Serialization

When you use XML Serialization, you must use interoperable, or interop, parameters and return types, as follows:

Interoperable Types

TYPE	DESCRIPTION
Primitives	`bool, byte( Java), sbyte (.NET)`, `byte[]`, `double`, `float`, `short`, `int`, `long`, `string`.
`Calendar` (Java) or `DateTime` (.NET)	Interoperability between Calendar and DateTime in a mixed Java/.NET setup does not support time zones. .NET's DateTime structure does not support the concept of timezone so that information is not maintained. You must send that information separately. The timezone itself is not preserved across serialization so all Calendar objects are created with local timezones. .NET has an additional limitation in that it assumes that all date time data is in its local timezone. Therefore, serialization of XML not in that local timezone produces an incorrect time. If you cannot determine the .NET deserializer's local timezone from Java to set in the Calendar, pass the Calendar data as a String instead.
Arrays	The type can be an array of any interop type.
User-Defined Types	User-defined types can be used, as long as they follow the standard “bean” pattern. For both languages, use interoperable types (including other user-defined types) for all data. For Java, all data must be Java Bean properties; that is, each must have public get/set methods. For .NET, all data must be public fields. When generating a proxy for this Service type, user-defined types result in generated classes. You can have other data and methods in the type, such as private non-interop fields, but they are ignored and not reflected in the generated class. Also, user-defined types must be concrete; abstract classes and interfaces are not allowed.

TYPE	DESCRIPTION
Data References (Java and .NET)	This type can be used as an argument, return type, or GridCache object, and is interoperable whether XML Serialization is on or off. When generating a proxy for a Service type using Data References, it does not include all methods of the Data References. Normally, the data in a Data Reference is stored and served in the file server on the GridServer component that created it. C++ Data References are not interoperable with Java or .NET. For more information, see Data References.

The following is an example of a Java Interop type:

```
public class Valuation {

    private java.util.Calendar valuationDate;

    private double value;

    private MarketData data;

    private String[] names;

    public Valuation() {}

    public java.util.Calendar getValuationDate() {

        return valuationDate;

    }

    public void setValuationDate(java.util.Calendar

        valuationDate) {

        this.valuationDate = valuationDate;

    }
```

```
    public double getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
    }
    public String[] getNames() {
        return names;
    }
    public String getNames(int index) {
        return names[index];
    }
    public void setNames(String[] names) {
        this.names = names;
    }
    public void setNames(int index, String name ) {
        this.names[index] = name;
    }
}
    public void setValuationDate(java.util.Calendar valuationDate)     {
        this.valuationDate = valuationDate;
    }
```

```
    public double getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
    }
}
```

The following is an example of a .NET Interop type:

```
[Serializable]
    public class Valuation {
        public DateTime valuationDate;
        public double value;
    }
```

R Interoperability

Before you create an R Service, you must decide which types of clients you intend to support. There are restrictions to the input and return types depending on the client that is used.

There are three main issues to address for each client type:

- Input and return value types
- `ServiceType` definition
- `init` method interface

Input/return Values and the functionInterface Setting

The `functionInterface` setting in the Service Type determines the conversion scheme that is applied to incoming input arguments and the outgoing return value.

There are three possible settings:

- `RAW` (default) - Input arguments and return values are not converted at all.
- `UNWRAP_RETURN` - Input arguments are not converted at all. However, if an R function returns a vector with a single element, the single element in the vector is returned instead. This is convenient for Java and .NET Drivers and a requirement for C++ Drivers. This setting has no effect if the returned value has more than one value.
- `NATIVE_R` - Input arguments and return values are native R types. Therefore they are not subject to any conversion.

Below are examples of how to create R Services depending on the client that has access to it.

Supporting C++ Clients

The C++ client is the most restrictive client to support.

Input and return type

C++ clients only support strings as input arguments and return values. Therefore, if your Service is accessed by a C++ client, all R functions exposed in the Service must only take strings and return a single string value.

ServiceType

Set `functionInterface` to `UNWRAP_RETURN` for C++ clients.

init method

If your Service has an init method, it must take at least one argument. For example:

```
init <- function(unused) {



}
```

The init method requires at least one argument even when `ServiceFactory::createService(const std::string &serviceName)` is used.

The init method can have multiple string arguments and the precise number of parameters must be provided as `initData` when creating a Service. Otherwise, Service initialization fails.

Supporting Java/.NET Clients

Java and .NET clients are less restrictive than the C++ client.

Input and return type

Java and .NET clients support their respective native types, and these types can be used as input arguments and return values.

ServiceType

`functionInterface` can be unset, or set to `UNWRAP_RETURN` depending your preference.

init method

The init method can have multiple arguments and the precise number of parameters must be provided as `initData` when creating a Service. Otherwise, Service initialization fails.

Supporting R Clients in R mode

R clients can access Services exposed using the C++ like Service interface and Services implemented in R and exposed as `NATIVE_R`.

R clients can call Services with the C++ like interface and Services exposed as `NATIVE_R` concurrently. Keep in mind that you cannot expose some functions with the C++ like interface and others as `NATIVE_R` in the same Service Type.

Input and return type

R clients can use native R types as arguments and return values. Refer to the list of supported types below.

ServiceType

`functionInterface` can be set to `NATIVE_R`.

init method

The `init` method can have multiple arguments and the precise number of parameters must be provided as `initData` when creating a Service. Otherwise, Service initialization fails.

Supported R Types

Supported R types are:

- `logical`

- `integer`
- `real`
- `raw`
- `complex`
- `nil` (`NULL` value)
- `string`
- `vector`

`vector` can contain any combination of the above types and `vector` itself.

Python Interoperability

You must carefully handle type conversion since the Python Driver wraps the C++ Driver and the data is being implicitly converted to string before being transmitted, and then it is implicitly converted from string in the Grid Library unless explicit conversion is used.

Maintaining State

To maintain state on a non-virtualized Service, you typically use a field or set of fields in your object to maintain that state. (For C++ or Command Services, state is saved in a slightly different manner.) Because a Service Session can be virtualized on a number of Engines, adjusting a field's value using a Service request adjusts only that value on the Engine that processed that request. Instead, you must declare the appropriate class method as a stateful method in the Service Type Registry and use the `updateState` method to guarantee that all Engines update the state. Register all methods used to update the state as such on the Service type, either as one of the `setStateMethod` s or `appendStateMethod` s.

When an Engine processes a Service request, it first processes all update state calls that it has not yet processed, in the order in which they were called on the Service instance. These calls are made prior to the execution of the request. The append value determines whether to make previous update calls. If append is false (a *set*), all previous update calls are ignored. If append is true, all calls starting from the last set call are performed. Typically, then, append calls update a subset of the state, whereas set calls refresh the entire state. If you intend your Service instance to be a long running state with frequent updates, use a set call on a regular basis so that Engines just coming online do not need to perform a long set of updates the first time they work on this Service instance.

In the case of a task retry when appending state, a task is appended until the correct update state required by the task is reached. Note that this does not occur when using multiplexed Engines — the task receives the most current state.

Initialization

Use the `initMethod`, one of the container binding fields defined above, to initialize the state on a Service that maintains state. You can also use it for other purposes, such as establishing a database connection. The `initMethod` is called with the initialization data the first time an Engine processes a request on a Service instance. It is also called prior to an `updateState` call if it has not already been called.

Cancellation

A request can be canceled for a number of reasons. The Admin API or Administration Tool can directly cancel requests. Canceling a Service Session cancels a request. If the `killCancelledTasks` option is true for this Service, the Engine process exits and the Engine restarts. (By default, the `killCancelledTasks` option is true, except for .NET Services, where it is false by default.) However, in many cases, it is not necessary to do so, and you would prefer to interrupt the calculation so that the Engine becomes immediately available.

In this case, set the `killCancelledTasks` option to false, implement a `cancelMethod`, and register it on the Service type. This method must interrupt any Service method that is in process. It is also possible to call `cancelMethod` after the Service method finishes processing, so the implementer must take this into account.

> **Note:**

> Use the `cancelMethod` to interrupt the task execution only. Do not use `cancelMethod` for cleanup of any sort. Use the `destroyMethod` for cleanup.

> A typical use case is a calculation that periodically checks a cancel flag. The cancel method would set that flag, interrupting the calculation.

Destruction

Often, a Service needs to perform cleanup operations on the Engine when the instance is destroyed, such as closing a database connection. If so, implement and register a `destroyMethod` on the Service type. This method is called whenever a Service instance is

destroyed. It is called on any active Service Sessions on an Engine whenever an Engine shuts down.

Service Instance Caching

Engines maintain a cache of all Engine Service Instances that are currently active on that Engine, set by the Engine Configuration. If an Engine is working on too many Sessions, Engine Service Instances can get dropped from the cache. When this happens, the `destroyMethod` is called, and work that began on that Service is discarded. If an Engine processes a subsequent request, it restarts working on that Service from scratch.

Invocation Variables

While you can implement a Service that is independent of DataSynapse libraries, on certain occasions you might need to refer to properties within the environment of the Engine. This can be accomplished without using additional code with variables that are retrieved in various ways depending on the type of Service:

- **Java** System properties
- **DynamicLibrary** Environment variables, with the same name as Java variables, except with dots replaced with underscores. You can also use symbolic constants provided by `DynamicLibraryFunctions.h`.
- **.NET** `System.AppDomain.CurrentDomain` data values; for example: `System.AppDomain.CurrentDomain.GetData("ds.ServiceSessionID")`
- **Command** Environment variables, with the same name as Java variables, except with the dots replaced with underscores.

The Engine provides the following variables:

Engine Variables

VARIABLE	DESCRIPTION
`ds.ServiceSessionID`	The unique identifier for the Service Session being invoked.
`ds.ServiceInvocationID`	A number uniquely identifying the invocation (task) of the Service instance.
`ds.ServiceCheckpointDir`	The absolute path to the directory that the Service uses for reading and writing of

VARIABLE	DESCRIPTION
	checkpoint data, if checkpointing is enabled for the Service.
`ds.ServiceInfo`	If this variable is set in an invocation, the value is displayed in the Administration Tool on completion of the invocation.
`ds.WorkDir`	The work directory for the Engine. This variable is set to the directory from which the Service is executed. By default, it is the absolute path to the Engine installation directory, where `machinename` is the name of the machine running the Engine, and `instance` is the number of the Engine instance. For example, a single Engine machine has a *`machinename`* `-0` directory; one with two Engine instances also has a *`machinename`* `-1` directory. Each Engine instance directory has a `log` directory containing Engine logs and a `tmp` directory. Note that the `tmp` directory is periodically deleted by the Engine. The **Temp File Time-to-Live (hours)**setting in each Engine Configuration controls the frequency with which the Engine cleans this directory.
`ds.DataDir`	The data directory for the Engine, which is the absolute path directory in which DDT (Direct Data Transfer) data is stored. The cleanup frequency is also controlled by the Engine Configuration.
`ds.GridLibraryPath`	A list of paths, the contents of which are absolute paths to the root directories of all the expanded and loaded Grid Libraries.

In addition, any environment variables available on the Engine are also available in the Engine Service Instance. Note that any environment variables use absolute paths.

Accessing Services

This section describes how to access and use Services with GridServer.

Services

You can use the GridServer API in Java, C++, .NET, COM, or Python to develop an application that accesses Services. The Java, C++, .NET, and COM Drivers contain API documentation describing how to do this. For example, when you use Java, refer to the Javadocs found in the GridServer Administration Tool, for the package `com.datasynapse.gridserver.client`. Use the `Service` class to access the Service either synchronously or asynchronously.

For examples of developing applications using the GridServer APIs, see the *GridServer Service-Oriented Integration Tutorial*.

Proxy Generation

Proxy Generation is the automatic generation of a client proxy class that mirrors the registered Service type.

Think of the Service as a binding to a virtualized Web Service that can process asynchronous requests in parallel. Additionally, because the proxy does not expose any DataSynapse classes, it provides a standards-compliant approach to integrating applications in a vendor non-specific way.

The following rules apply to the generated proxy:

- The use of the proxy class is completely independent of the DataSynapse API. Client code that uses the proxy class does not need to import or reference any DataSynapse classes.
- If the Service has an `initMethod`, the proxy constructor takes any arguments to that method.
- All Service methods produce synchronous and asynchronous versions of the method on the proxy.
- Each update method has a corresponding update method on the proxy.
- The `cancelMethod` and `destroyMethod` are called implicitly and thus do not generate methods on the proxy.
- The `targetPackage` field identifies the package (in Java) or namespace (in .NET) in which to place the generated classes. Its default value is the name of the Service.
- If you use xmlSerialization, classes are generated for all non-primitive types, which must be interop types. If you do not use xmlSerialization, classes can be any serializable type, they are not generated, and the client must have access to those same classes (by a JAR/Assembly.)

- When generating proxies, a Java `Calendar` object is represented by a `DateTime` object in a .NET proxy, and vice-versa.

The proxy is generated using the **Services > Services >****Service Types**page. The proxy is generated on an Engine, so successful proxy generation requires an available idle Engine.

Service Options

Each Service has an Options object that contains configuration parameters and settings. For example, some commonly used options include `PRIORITY` and `GRID_LIBRARY`. To see the options for the `Options` object, refer to the API reference documentation.

Service Options can be set in two ways: on the **Service Types**page, or when you create the Service Session with the client. If an option is set in the registry, the client cannot override it. If it is left as `[not set]` in the registry, and it is not set by the client, Service Options has the default value.

Service Invocation Context

The `ServiceInvocationContext` class provides an interface for interacting with an invocation, such as getting the session and task IDs, while it is running on an Engine. This is an alternative to using, for example, the system properties when running a Java Service. Using this class enables immediate updating of invocation information. In contrast, setting the `INVOCATION_INFO` system property only updates at the end of the invocation.

The `ServiceInvocationContext` object can be reused; the method calls always apply to the currently executing Service Session and invocation. Make all method calls by a `service`, `update`, or `init` method; if not, the method call might throw an `IllegalStateException` or return invalid data. Note that you cannot call this method from a different thread; it fails if it is not called from the main thread.

Setting Task Description

An example use case of the `ServiceInvocationContext` class is how to change the task info field.

You can use the `ServiceInvocationContext.updateInfo()` method on the Engine to set the info, and can be updated a number of times while the task is running to provide real-time info on the task state.

Shared Services

A *Shared Service* is a Service instance that multiple Clients executing on different processes or machines can attach to. State is maintained across the Service instance, and there is only one instance at any given time of a particular Shared Service running on a Broker. For example, you might use a Shared Service when GridServer Service invocation requests are delivered on an Enterprise Service Bus to a middle-tier client. If this middle-tier client is load-balanced across multiple systems, using a Shared Service lets the Broker reaggregate these related Service invocations.

Creating a Shared Service

To create a Shared Service, specify the `SHARED_SERVICE_NAME` option when creating the Service. Once the Shared Service exists, any client of the same type attempting to create a Service with the same name and the same `SHARED_SERVICE_NAME` attaches to the already existing Service. All clients sharing the instance also share the same Service ID.

Limitations to Shared Services

There are certain limitations to Shared Services, listed below:

- If there is an init method, it cannot take any arguments.
- Service options are set when the Service Instance is created initially; you cannot override these Service options on subsequent attaches.
- Clients that share Services must be the same type. For example, a Java Driver cannot share a Service created with CPPDriver.

> **Note:**
>
> **Never**use the same Shared Service from within the same process, or launch multiple Drivers with the same data/log directories. Shared Services are not designed for such situations and there is no benefit or reason to do so. Doing so can result in premature task data deletion and poor performance.

Ending a Shared Service

To cancel an entire Shared Service, use the Administration tool (at **Services > Servides > Service Session Admin**) or the Web Service Admin interface. When a client cancels a Shared Service instance, it cancels only the tasks it submitted; other clients' tasks continue. The Shared Service becomes inactive when the last client detaches from the Service. The inactive Service closes after

waiting for the amount of time specified by `SHARED_SERVICE_INACTIVITY_KEEP_ALIVE_TIME`. If this variable is not set or has a value of 0, the Service closes immediately.

Shared Services and Failover

Do not depend upon Engine state for a Shared Service being maintained through failover. Engine update information can reside on several Drivers running the Shared Service. If, after failover, all of these Drivers are not available or not running the Service, Engine state is indeterminate.

Broker Spanning Services

A normal Service runs on a single Broker and is available to the Engines connected to that Broker. Using Broker Spanning, a single Service can take advantage of all Engines on an entire grid across all of the grid's Brokers. When Broker Spanning is enabled on a Driver, the Driver concurrently maintains connections to a user-defined set of Brokers, and tasks for Services are submitted to all of those Brokers.

Enabling Broker Spanning on a Driver

Broker Spanning is enabled at the Driver level. When enabled, all Services are panned over all Brokers. To enable Broker Spanning on a Driver, set `DSBrokerList` to a semicolon-delimited string of Broker names for all the Brokers you want to span.

For example:

```
DSBrokerList=London1;London2;Cambridge
```

Note that you must allow the Driver on all Brokers in the list.

You can also set `DSBrokerList` to the wildcard character of `*`. This sets the list to all Brokers on which the Driver has the Execute Services permission. This includes offline Brokers.

When Broker Spanning is enabled, the Driver submits an independent Service for each of the Brokers specified in `DSBrokerList`. The Services submitted to each of the Brokers are separate entities, but for the convenience of the user, have the same Service ID. This allows the user to reference the Services easily across the Brokers when using the Dashboard. While Service IDs for the Broker Spanning Services are the same across all spanned Brokers, the Task IDs for the spanned Service are unique across all spanned Brokers and the Task ID is maintained if the task is moved across spanned Brokers.

Admin API Usage

For the purposes of Broker Spanning, the Admin API provides a way to specify which Manager the Admin API is acting on. `AdminManager.setManager()` takes the name of the Manager it acts on as a string parameter. Admin API components retrieved act only on the Manager most recently specified by `AdminManager.setManager()`. To perform Admin API operations on multiple Managers at the same time, prepare collections of Admin API component objects retrieved after each `AdminManager.setManager()` call. For example, if you want to get all `EngineInfo` s for all spanned Managers, call `AdminManager.setManager()` followed by a call to `AdminManager.getEngineAdmin()` for the Manager last specified by `AdminManager.setManager()`. This must be done for each of the Managers. Each of the `EngineAdmins` can now be used to call `EngineAdmin.getAllEngineInfo()` to retrieve all the `EngineInfos` associated with that `EngineAdmin's` Manager. See The Admin API for more information.

Scheduling and Task Expiration

When Broker Spanning is enabled, the Driver continuously attempts to maintain connections to all Brokers. It assigns tasks to Brokers using a simple algorithm that attempts to balance the workload. If a connection times out on any Broker, the Driver does Driver-side expiration and resubmits any outstanding tasks to the remaining Brokers. In the event of a Broker failure, long-running tasks continue to run on Engines; when they log in to the Director, they are directed to whichever Broker that has ended up with the expired task.

After submitting a large number of tasks, some Brokers might contain tasks in the queue to execute whereas other Brokers might not. When Task Expiration is enabled, tasks expire on the Broker if they are not scheduled to any Engine after a specified period of time after submission. Task Expiration alleviates the problem of unbalanced Broker queues. Upon expiration, tasks are automatically assigned to the next appropriate Broker. Task Expiration is available for Broker Spanning Services only.

To activate Task Expiration, set the following Service options:

- `PENDING_TASK_TIMEOUT` — To enable Task Expiration, set this to a value indicating the number of minutes to wait before reassigning tasks to another Broker.
- `TASK_EXPIRATION_START_OFFSET` — Optionally enables you to delay the start of Task Expiration by a specified number of minutes. This offset represents a one-time delay in starting the timer to check for Task Expiration. For example, if the value of `PENDING_TASK_TIMEOUT` is set to 5 and `TASK_EXPIRATION_START_OFFSET` is set to 60, the first task to be expired is 65 minutes after the Service was submitted to the Broker. Additional task expiration would happen

every 5 minutes after the first expiration event. The Broker first waited for `TASK_EXPIRATION_START_OFFSET` to pass (60 minutes) before starting the `PENDING_TASK_TIMEOUT` timer (5 minutes) for a total of 65 minutes. The default value for `TASK_EXPIRATION_START_OFFSET` is 0.

You can also tune Task Expiration on the Broker by specifying the following values in the Administration Tool at **Admin > System Config > Manager Configuration > Services**:

- **Minimum Tasks to Expire**— You can require that a certain number of tasks expire regardless of the Percentage of Tasks to Expire (Given that there are eligible tasks to expire).
- **Percentage of Tasks To Expire**— You can expire only a percentage of tasks of those eligible for expiration every minute.
- **Idle Broker Only Task Expiration Enabled**— When enabled (the default), this setting prevents tasks from expiring if there is not an idle Broker available.

Administration

Broker Spanning Services can be viewed and administered in the GridServer Administration Tool, at **Dashboard > Spanned Services**. This page is only viewable on Directors. By default, this page shows no Services; you must first perform a search using the Search tool. Administrators can dynamically change Pending Task Timeout and Task Expiration Start Offset on each Service from this page.

Broker Spanning Service Limitations

Broker Spanning Service limitations are as follows:

- Shared Services is not supported. Shared Service functionality conflicts with Broker Spanning Services.
- GridCache is not supported. Since a Service’s cache is local to a Broker, the tasks from other Brokers cannot access each other’s caches.
- Task Dependencies are not supported. Task Dependency is Broker-scope and therefore cannot work across Brokers.
- Service dependencies are not supported. Service Dependency is Broker-scope and therefore cannot work across Brokers.
- AutoPack is not supported.
- PDriver is not supported.

- The number of priority levels, which is set at **Admin > System Admin > Manager Configuration > Services > Scheduling**, must be the same on all Brokers. If you change this value on a Broker, you must restart the Broker.
- Engine Timeout Minutes > 0 is not supported. That is, if an Engine logs off while working on a Task, it does not continue to work on that while attempting to log back in to a Broker.

Service Groups

Service Sessions can be collected together in a group to aid in administrative tasks. A convenience class called `ServiceGroup` is provided in the API, which enables you to create a Service Group and later create new Service Sessions within the Service Group. Each new Service Session created within a Service Group is automatically assigned `Description.SERVICE_GROUP_ID`, a generated random unique ID for that group.

To view and maintain Service Groups, select **Services > Services > Service Group Admin**. The Service Group Admin page enables you to take actions on an entire group of Services at once, similar to the way you can act on Services on the **Services > Services > Service Session Admin**Page. For example, you could cancel a Service Group, which would cancel all of the Service Sessions within that Service Group.

Data References

Data References are a convenient programming interface for passing lightweight references to data across the network. A Grid client or Service can create Data References, pass them over the network, but leave the data where it created the original Data Reference. If any Grid client or Grid node actually needs the data, it can de-reference the object and the data is automatically downloaded from the original source.

You can use this abstraction for generalizing Grid workflows. A Grid client can receive the results of a particular Service as a reference, and then send another request to the Grid with that reference. The GridServer Engine that services the request de-references the data object, loading it from the original Grid node that produced the data. This is equivalent to passing pointers across the network.

A `DataReference` is an Object that you can pass interoperably if you use it as an argument, return type, or a GridCache object. Note that to work interoperably, it must be the actual object passed; it can’t be part of another object.

To create a `DataReference`, use a `DataReferenceFactory`. You cannot create Data References with a null source. To retrieve the actual data, use the `fetch` methods. The data is

not cached after a `fetch`, that is, each fetch retrieves data. After a reference expires, a `fetch throws` a FileNotFound exception. A downed Client throws a Connect exception, although in some cases, refused connections are due to other causes, such as socket backlog limitations.

When an object is transmitted over the network, it is serialized on the sending side and then reconstituted completely in memory on the receiving side. The larger the object, the more memory it occupies. In situations with large `DataReference` objects, you must use `DataReferenceOutputStream` instead. Streaming the data can reduce the memory that would otherwise be necessary to reconstitute it in its entirety. As an added benefit, streaming also saves the time it would have taken to reconstruct the entire object.

C++ Data References

Because there is no inherent serialization support in C++, functionality is added to convert a Data Reference to a `byte[]` (and the reverse) so that it can be sent to another client and used by that client.

Because reflection is not available in C++, object Data References are not available from the C++ API.

Python Data References

As a result of Python wrapping the C++ Driver, object Data References are not available from the Python API.

Service Collection

By default, the Driver Service Instance collects results as soon as they are available. In some cases, you want to collect results at a later time, or never collect them. Service collection can be defined with the Service Option `COLLECTION_TYPE`. Possible values for `COLLECTION_TYPE` include `IMMEDIATELY`, `LATER`, `NEVER`, and `AFTER_SUBMIT`.Values used with `COLLECTION_TYPE` are as follows:

Service Collection Values

VALUE	DESCRIPTION
`AFTER_SUBMIT`	Collection does not begin until all tasks have been submitted. See Collect After Submit for more information. This enables optimal submission without network competition from

VALUE	DESCRIPTION
	the collection when the Service follows the “submit all, collect results” pattern.
`IMMEDIATELY`	Task outputs are collected as soon as they are ready. This is the default.
`LATER`	Task outputs are not collected in this Service. Another Service collects them later. See Deferred Collection (Collect Later) for more information. This is not available from a Service proxy.
`NEVER`	Task outputs are not collected. See No Collection (Collect Never) for more information. This can be used, for example, in the case where a Service write data directly to a database. This is not available from a Service proxy.

The `LATER` and `NEVER` collection modes are not for long-running Services. Use them only for batch submissions that finish quickly. If you use them for Services with indefinite duration, there is no way to clean up the inputs.

Collect After Submit

A common Service pattern is to submit all of your requests, then wait for the results. When using this pattern, the `AFTER_SUBMIT` collection mode must be used. This enables optimal submission without network competition from the collection.

When this mode is used, the collection of the results begins when either `Service.waitUntilInactive(long)` or `Service.destroyWhenInactive()` is called.

Also, when using this mode or even when using some hybrid pattern that submits many requests at a time, `Options.INVOCATIONS_PER_MESSAGE` must **always**be set greater than one, to speed submission time.

Calling `Service.execute(String methodname, Object data)` does not work with this collection type; an instance of `ServiceException` is thrown.

The following is an example of using the `AFTER_SUBMIT` collection mode:

```
Properties props = new Properties();

props.setProperty(Options.COLLECTION_TYPE,

                  Options.Collection.AFTER_SUBMIT);

props.setProperty(Options.INVOCATIONS_PER_MESSAGE, "20");

Service s = ServiceFactory.getInstance().createService(serviceType, null,
props, null);

// submit all the work

for (int i = 0; i < numInputs; i++) {

    s.submit("myMethod", getData(i), handler);

    }

s.waitUntilInactive();
```

Fault Tolerance

In the event of a Driver failure, the Broker automatically cancels the Service after the interval specified by the Client Timeout setting. It is possible to collect tasks from a failed client application for Services of type `Collection.AFTER_SUBMIT` and `Collection.IMMEDIATELY` if cancelslation has not yet occurred. To do this, you must collect and pass in the Driver session ID, in the same fashion as the collection of a `Collection.LATER` Service, as described in Deferred Collection (Collect Later).

Note that for fault tolerance to work, the collecting Driver must be the same platform as the original submitting Driver. That is, if the Service was created with Java Driver, it must also be collected with Java Driver.

Deferred Collection (Collect Later)

The `LATER` mode indicates that the client can submit and update data to the Service Session. It does not collect the results, however, as another instance attaches to the Service Session to collect the results. None of the results are removed from the Service Session until it is destroyed by the collector. The `LATER` mode cannot be used from a Service proxy.

There are two reasons to use this method:

1. The architecture requires different applications for submission and results processing.
2. To recover from a failure in the application that embeds the Driver. Since results are not removed until the Session is destroyed, if the application undergoes a failure it can recollect the results when it restarts.

Deferred collection Services require that the submitting Driver call destroy on the Service to indicate that submission is complete. If you are using the submitting Driver in such a way that it exits after submitting the tasks and calling destroy, do not call `System.exit` or `exit` from the `ServiceLifecycleHandler`, as this stops the destroy message from getting to the Broker. Also note that if you are exiting the submitting Driver immediately, you must set `DirectDataTransfer` to false in the `driver.properties` file.

After creating a Service with the deferred collection, use `ServiceFactory.getService()` to retrieve results. After you collect all results, call destroy to indicate to the Broker:

- that the instance has collected all outputs and
- to destroy the Session

You can create multiple collectors, but keep in mind that if a collector calls `destroy()`, the Service is destroyed and no other collectors can finish collecting outputs.

The following is an example of how to use the `LATER` mode with recovery:

```
// Creates a new Session and submits requests to the Session

// @param serviceType The type

// @param methods The list of methods

// @param args The list of arguments

// @return The id of the Session

// @throws Exception on error.

private String submitService(String serviceType, String methods[],
Object[][] args) throws Exception {

    // create the session as a Collection.LATER type

    Properties props = new Properties();
```

```
    props.setProperty(Options.COLLECTION_TYPE, Options.Collection.LATER);

    Service cs = ServiceFactory.getInstance().createService(serviceType,

        null, props, null);

    // Submit all requests.

    // Note that the handler must be null because this Instance cannot
collect.

    for (int i = 0; i < args.length; i++) {

        cs.submit(methods[i], args[i], null);

        }

    String id = cs.getId();

    // destroy to indicate that submission is complete, and to free local

    // resources

    cs.destroy();

    // save this ID to a file, for recovery purposes

    saveServiceForRecovery(id);

    return id;

}

// Starts collection of results from a Collection.LATER Session

// @param id The id of the session

// @param handler The invocation handler

// @throws Exception on error.

private void collectService(final String id, ServiceInvocationHandler
handler) throws Exception {
```

```
    // create a handler that removes this Service ID from the list in the
file

    // when it is finished

    ServiceLifecycleHandler slc = new ServiceLifecycleHandler() {

        public void destroyed() {

            removeServiceFromRecovery(id);

        }

        public void destroyed(ServiceException e) {

            removeServiceFromRecovery(id);

        }

    };

    // get an instance of the session, which starts collecting results

    Service cs = ServiceFactory.getInstance().getService(id, handler,
slc);

    // set the Service to be destroyed when it finishes collecting all
output

    cs.destroyWhenInactive();

}

// Runs a Service by first creating a Collection.LATER Session, submitting
all

// requests, then getting the collection instance to collect the results.

// @param serviceType The type

// @param methods The list of methods

// @param args The list of arguments
```

```
// @param handler The invocation handler

// @throws Exception on error.

private void runService(String serviceType, String methods[], Object[][]
args, ServiceInvocationHandler handler) throws Exception{

    String id = submitService(serviceType, methods, args);

    collectService(id, handler);

}

// Recovers from an application failure by starting collection of Sessions
that

// did not complete collection prior to failure

// @param handler The invocation handler

// @throws Exception on error.

//

private void recoverAll(ServiceInvocationHandler handler) throws Exception
{

    String[] recovered = getAllRecoveryServices();

    for (int i = 0; i < recovered.length; i++) {

        collectService(recovered[i], handler);

    }

}
```

Deferred Collection Failover

While other Sessions manage failover by resubmitting all outstanding tasks to a Failover Broker, in this case the submitting Session has already been destroyed, and the Driver might even be offline. So, to handle failover in this case, you must do the following:

1. Every standard Broker must have one Failover Broker associated with it.
2. Configure a shared filesystem that can be accessed from each Broker pair.
3. Go to **Admin > System Admin > Manager Configuration > Services**, and under the **DataTransfer**heading, change the **DataTransfer****Base Directory**to the location of the shared file system for each pair. Each pair must have its own directory.
4. If the submitting Driver does not use an external web server and it disconnects after submission, disable DDT. The data is then served by the Broker from the shared file system.
5. Since the collecting Driver cannot auto-resubmit when the results cannot be collected from a Daemon that is offline, it's recommended that DDT is also disabled on Engines.
6. The submitting and collecting Drivers must be allowed to log in to both Brokers in the pair.

If a Broker fails while the Service is running, the Failover Broker takes over by recreating the task queues. If the original Broker comes back up, the Failover relinquishes control and the original recreates the task queues.

When the collecting Driver starts to collect, if the original is down, it collects results from the Failover. If that then comes up, it migrates to the original and continues.

No Collection (Collect Never)

The `NEVER` collection mode enables a Service to submit tasks and not collect them. Such a Service, for example, writes results to a database. Services created with `NEVER` collection can only submit and update. Calls to execute throw an Exception. This collection mode is not available when using a Service proxy. To create a `NEVER` collection Service, set the `CollectionType` option to `NEVER`.

Calling destroy releases resources locally on the Driver, and indicates that the Instance is finished with submission. If the Driver is shut down and times out, the session is considered to be done submitting, just as if the Driver called destroy. After the session finishes submitting due to one of the two prior events, and all tasks complete or fail, the session automatically closes.

No Collection Failover

There are no special issues with failover on No Collection Services, and as many Brokers as you want can be used to recover any running Services. You would, however, need to ensure that any database to which you are sending results has its own failover or redundancy configuration in case of failure.

Engine Pinning

Engine Pinning enables a Service Session to specify that once an Engine has worked on a Service Session, it cannot work on any other Service Sessions as long as that session is in progress. This enables you to quickly replicate 1-to-1 architectures, or if for legacy reasons or lack of availability to source code to maintain massive amounts of expensive-to-replay state on an Engine.

Typically, you use Engine Pinning in conjunction with Max Engines to limit one or a set of Engines to work solely on a Service.

Engine Pinning is available for any type of Service. It is exposed as two Engine-side methods, `pinToService()` and `unpinFromService()`. If an Engine calls `pinToService()` while working on a task for a Service, it is marked as pinned to the session when it completes the task, regardless of whether the task succeeded. From this point, the Engine takes tasks only from this Service. Once the session finishes, or when the Engine calls `unpinFromService()`, the Engine is no longer pinned and can work on other Services.

Engines that are pinned are unpinned automatically in the following circumstances:

- If the Broker loses its connection with the Engine for any reason (such as loss of heartbeat)
- If the Engine process ends for any reason
- If an Engine performs a soft log off due to Engine Sharing
- If a task fails due to an error in DataSynapse code (such as reading input)
- If the session is destroyed

Exceptions issued by user code that result in task failure do not cause an Engine to be unpinned unless the exception specifies Engine restart, in which case the above requirement applies.

You can configure Engine balancing so pinned Engines are always reported as busy by Brokers. This prevents the Engine from being shared with other Brokers. This can be configured at **Admin > System Admin > Manager Configuration > Engines and Clients**, under the **Engine Balancing**heading, with the **Treat Pinned Engines as Busy**parameter. This is false by default.

You can implement other pin/unpin strategies—for instance, unpinning after a certain amount of idle time, or when the Broker queue is empty—as a separate Engine thread that polls for a given condition and unpins as necessary.

Running a Driver from an Engine Service

It’s possible to run a recursive Service, which is a Service that runs a Driver on the Engine to call a Service. To do this, you must deploy the Driver packaged in a Grid Library.

To run a Driver from an Engine Service:

1. In the GridServer Administration Tool, go to the top navigation bar, and click the **Downloads**icon.
2. Click the SDK for your platform to download it.
3. Package the `driver.properties` file in a Grid Library. (For more information about creating Grid Libraries, see Creating Grid Libraries.) In the `grid-library.xml`, set the environment variable `DSDRIVER_DIR` to the value `$ds.GridLibraryRoot$`. Also, set the `jar-path` element to `$ds.GridLibraryRoot$`. The `grid-library.xml` must look similar to the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<grid-library>
    <grid-library-name>
        driver-properties
    </grid-library-name>
   <grid-library-version>7.1</grid-library-version>
   <environment-variables>
        <property>
            <name>DSDRIVER_DIR</name>
            <value>$ds.GridLibraryRoot$</value>
        </property>
    </environment-variables>
    <jar-path>
        <pathelement>
            $ds.GridLibraryRoot$
```

```
            </pathelement>
        </jar-path>
    </grid-library>
```

4. Create a Grid Library that contains all of the DLLs or JAR required to run the Driver. For example, to run CPPDriver for VC14, you would need to create a Grid Library that contained all of the DLLs from the `cppdriver/bin/vc14` directory of the SDK in a `bin/vc14` directory inside the Grid Library. Make this Grid Library dependent on the `driver-properties` Grid Library, and set the `lib-path` to the location of the DLLs. The `grid-library.xml` must look similar to the following code:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <grid-library os="win64">
        <grid-library-name>
            cppdriver-win64-vc14
        </grid-library-name>
        <grid-library-version>7.1</grid-library-version>
        <dependency>
           <grid-library-name>
                driver-properties
            </grid-library-name>
        </dependency>
        <lib-path>
            <pathelement>.\bin\vc14</pathelement>
        </lib-path>
```

```
</grid-library>
```

5. Upload and deploy both Grid Libraries on the **Services > Services > Grid Libraries**page.

PDriver

The *PDriver*, or Parametric Job Driver, can execute command-line programs as a parallel processing service using the GridServer environment. This driver enables you to run a single program on several Engines and return the results to a central location, without writing a program in Java, C++, or .NET.

Overview

The *PDriver*, or Parametric Job Driver, can execute command-line programs as a parallel processing service using the GridServer environment. This driver enables you to run a single program on several Engines and return the results to a central location, without writing a program in Java, C++, or .NET.

PDriver achieves parallelism by running the same program on Engines several times with different parameters. A script defines how these parameters change. For example, a distributed search mechanism using the `grep` command can search a network-attached file system, giving each task in the Service a different directory or piece of the file system to search. Scripts can iteratively change the value of variables that are passed to successive tasks as parameters, step through a range of numbers, and use each value as a parameter for each task that is created, or define variables containing lists of parameters.

PDriver uses its scripting language, called *PDS*, to define Services. You can also use these scripts to set options for a PDriver Service, such as remote logging and exit-code checking.

Installing PDriver

PDriver comes with the GridServer SDK. For installation details, see The Parametric Job Driver (PDriver).

To use PDriver with SSL, you must configure your Manager to use SSL for all connections. For directions, see the *TIBCO GridServer® Installation*.

Resource Deployment

If you plan to run any custom executables using PDriver, deploy them to Engines using GridServer’s application resource deployment feature, described in the *TIBCO GridServer®*

Administration. Deploy resources in the `deploy/resources/`*platform*`/lib` directory or in another directory referenced by the `execute` command.

PDriver Commands

Use the following commands to run, batch, and cancel PDriver Services on UNIX and Windows systems.

PDriver Commands

COMMAND	DESCRIPTION
`pdriver`	Starts PDriver, and runs a PDS script.
`bsub`	Starts a one-time Service batch submission.
`bcoll`	Collect a batch job on a one-time basis.
`bstatus`	Lists status on pending batch jobs.
`bcancel`	Cancels a pending batch job.

The pdriver Command

Start PDriver with the `pdriver` command:

```
pdriver [ -bsub | -bcoll batchid | -parallel] [-RA] [-noprompt] [-user
driverusername] [-pass driveruserpassword] [-domain windowsdomain] script
```

The `pdriver` command runs the script specified with *script*. The following table describes the arguments you can use:

PDriver Command Arguments

ARGUMENT	DESCRIPTION
`bsub` and `bcoll`	Use `bsub` and `bcoll` for batch submission and collection. Use them to • Submit long-running Services without tying up the Driver. Submitting a Service

ARGUMENT	DESCRIPTION
	using `bsub` creates a Batch Execution with a Batch ID that you can use to run the Service unattended and collect the outputs later. The Batch Executions created are of the same type as those from the Batch scheduling facility. For more information about Batches, see the *TIBCO GridServer® Administration*. • Run multiple tasks and or prejob/postjob tasks that the PDS script defines.
`parallel`	When you run a single PDS that contains multiple job blocks, this option runs the blocks in parallel.
`RA`	`RA` specifies a Run-as job—a job that runs as the current user and with the current Windows domain. PDriver prompts for a password and passes it with the user and domain for authentication. If you run PDriver on a UNIX machine, you must use the `domain` argument to pass a domain, as shown below. For more information about Run-As, including how to configure your Engines, the *TIBCO GridServer® Administration*.
`noprompt`	`noprompt` turns off the password prompt and sends no password when authenticating for Run-as; this is useful only when you use the `RA` argument and Service RA authentication is disabled on the Broker.
`user` *name* and `pass` *password*	When you use the `RA` argument and run PDriver on a Windows machine, you can use

ARGUMENT	DESCRIPTION
	`user` and `pass` to pass a Driver user and password (as defined in the GridServer Administration Tool) to PDriver.
`-domain <windowsdomain>`	When you are using the RA argument and running PDriver on a UNIX machine, you can use the `domain` argument to specify a win32 domain for Windows Engines to use when authenticating.

The bsub Command

You can also run a one-time Service batch submission with the `bsub` command:

```
bsub [ -name name ] [ -priority val ] [ -disc setting ] [ -mail address ]
[ -stdin file ] [ -stdout file ] [ -stderr file ] [ -nfs ] [-RA] [-domain
windowsdomain] [-noprompt] app [args...]
```

This submits a single Service, defined by *app* `[args...]`, to be scheduled and run on the Grid. The arguments are as follows:

bsub Command Arguments

ARGUMENT	DESCRIPTION
`name`	Name of the Service.
`priority`	Priority of the Service.
`disc`	Indicates a *name**comparator**value* discriminator expression that must be satisfied to assign a Service or task to a node. For example, valid settings include `os==win32` or `cpuNo>=4`. You can use this switch multiple times. Put quotes around parameters containing a greater-than or less-than symbol.
`mail`	Email address to send a confirmation message to when the Service is completed.

ARGUMENT	DESCRIPTION
`stdin`	File to use as input for the Service. This setting is variable based on the `-nfs` switch, described below.
`stdout`	File to use as output when `-nfs` is enabled.
`stderr`	File to use as error output when `-nfs` is enabled.
`nfs`	Indicates whether inputs and outputs are available on a shared file system or whether you must stage them on the Manager. When enabled, `-stdin`, `-stdout`, and `-stderr` function as absolute locators for these files. When not enabled, `-stdout` and `-stderr` are ignored, and files designated as *jobname*`.out` and *jobname*`.err` are placed in the staging directory. When not enabled, `-stdin` refers to a file on the Driver file system which is to be staged on the Manager and retrieved by the Engine performing the Service.
`RA`	Runs the job on Engines as the current desktop user with the current Windows domain. PDriver first prompts for a password.
`domain`	Lets you specify a Windows Domain for Windows Engines to use when authenticating. This applies only when running PDriver from a UNIX machine.
`noprompt`	`noprompt` turns off the password prompt; this is useful only when Service RA auth is disabled on the Broker.
`<app> [args...]`	The application to run, and any optional arguments. The application is a binary or script on the Engine and found in the Engine's path. `args` are any arguments passed to the application.

When you submit the Service, a batch ID is reported to the console. Use this ID when collecting outputs (when using the staging directory for input and output rather than NFS mounts.)

The bcoll Command

To collect batch jobs on a one-time basis, use the `bcoll` command:

```
bcoll batchid
```

This is a convenient utility for retrieving the *jobname* `.out` and *jobname* `.err` files generated by `bsub` with `nfs` mode off. The argument is the batch ID indicated when `bsub` is finished submitting.

To check the status of a Batch Job, you must be running PDriver with a Driver user that has the appropriate Security Role on the Manager. To do this, create a Driver user mapped to a role with access to the **Batch Admin View**feature (such as the default Manage role), and set `DSUsername` and `DSPassword` in the `driver.properties` file to this username.

The bstatus Command

To get status on batch jobs, use the `bstatus` command:

```
bstatus [ -jobs ] [ -engines ] [ -stagedir id ] [ -canceljob id ] [ -
batches]
```

`bstatus` accepts the following arguments:

bstatus Arguments

ARGUMENT	DESCRIPTION
`jobs`	Lists all jobs on the Manager. This does not include pending batch jobs. The output displays name, jobid, tasks, priority, and status for each job. If there are no jobs, only the four rows of heading lines are returned.
`engines`	Lists all Engines connected to the Manager. The output displays name, ID, OS, and status of each Engine. If there are no jobs, only the four rows of heading lines are returned.

ARGUMENT	DESCRIPTION
`stagedir`	Lists all files living on the staging directory for the given job ID or batchjob ID.
`canceljob`	Cancels a job. This only works for job IDs, not batch IDs, and the Driver must have appropriate permissions for this operation.
`batches`	Lists all Batch jobs on the Manager and their status. The output displays name, batchid, and status of each Batch. If there are no batches, no output is returned.

The `bstatus` command prints its output interspersed with Manager log output on `stderr`. To view only the `bstatus` output on a UNIX system, you can redirect `stderr` to null, for example, with `bstatus -jobs 2>/dev/null`. Windows users can use the syntax `bstatus -jobs 2>&0`.

To check the status of a Batch Job, you must be running PDriver with a Driver user that has the appropriate Security Role on the Manager. To do this, create a Driver user mapped to a role with access to the **Batch Admin View**feature (such as the default Manage role), and set `DSUsername` and `DSPassword` in the `driver.properties` file to this username.

The bcancel Command

To cancel a batch job pending on the Manager, use `bcancel`:

```
bcancel batchid
```

The argument is the batch ID returned by `bsub` or `pdriver` in `bsub` mode.

There is an important distinction between batch IDs and Service IDs. A batch ID refers to a Service that is pending execution in the batch queue but has not been handed over to the scheduler. Once that Service is scheduled, a Service with a separate Service ID is launched. The `-canceljob` switch in `bstatus` only works for Service IDs. To remove a pending batch job, `bcancel` must be used. This is unavoidable since running Services and pending batch jobs are separate entities and are tracked differently. Likewise, to perform output collection and to see the contents of the staging directory, the batch ID must be used. This is due to the fact that at batch submission, only the batch ID is known since the Service ID is not generated until run-time. Therefore the batch ID is used as a key for the staging directory.

About PDS Scripts

PDriver uses scripts written in the PDS syntax to define how a Service operates. Aside from defining what programs are run during a Service, PDS scripts enable you to define what happens before and after a task or Service. It also enables you to schedule Services to run in the future, add conditional structure to a Service, and pass custom parameters to a Service or task.

The PDriver script (hereafter referred to as PDS) language enables the expression of distributed computations that are composed of executable programs. It is designed so that typical computations are easy to describe, while providing for advanced features such as conditional execution, iteration, scheduling, and discriminators.

This section is a reference for the PDS language. For sample PDriver scripts, see the `examples/pdriver` directory of the GridServer SDK.

PDS Basics

A single PDS file corresponds to a single GridServer Service. The computation represented by the Service usually does the following:

1. Split up the input data into several pieces, one for each task.
2. Run the tasks in parallel on the Engines.
3. Collect and combine the results.

If the data is too large to be passed as command-line arguments to the program running on the Engine, then place it into files. You can locate these files in a shared directory, or copy them to the Engine and copy back the result files.

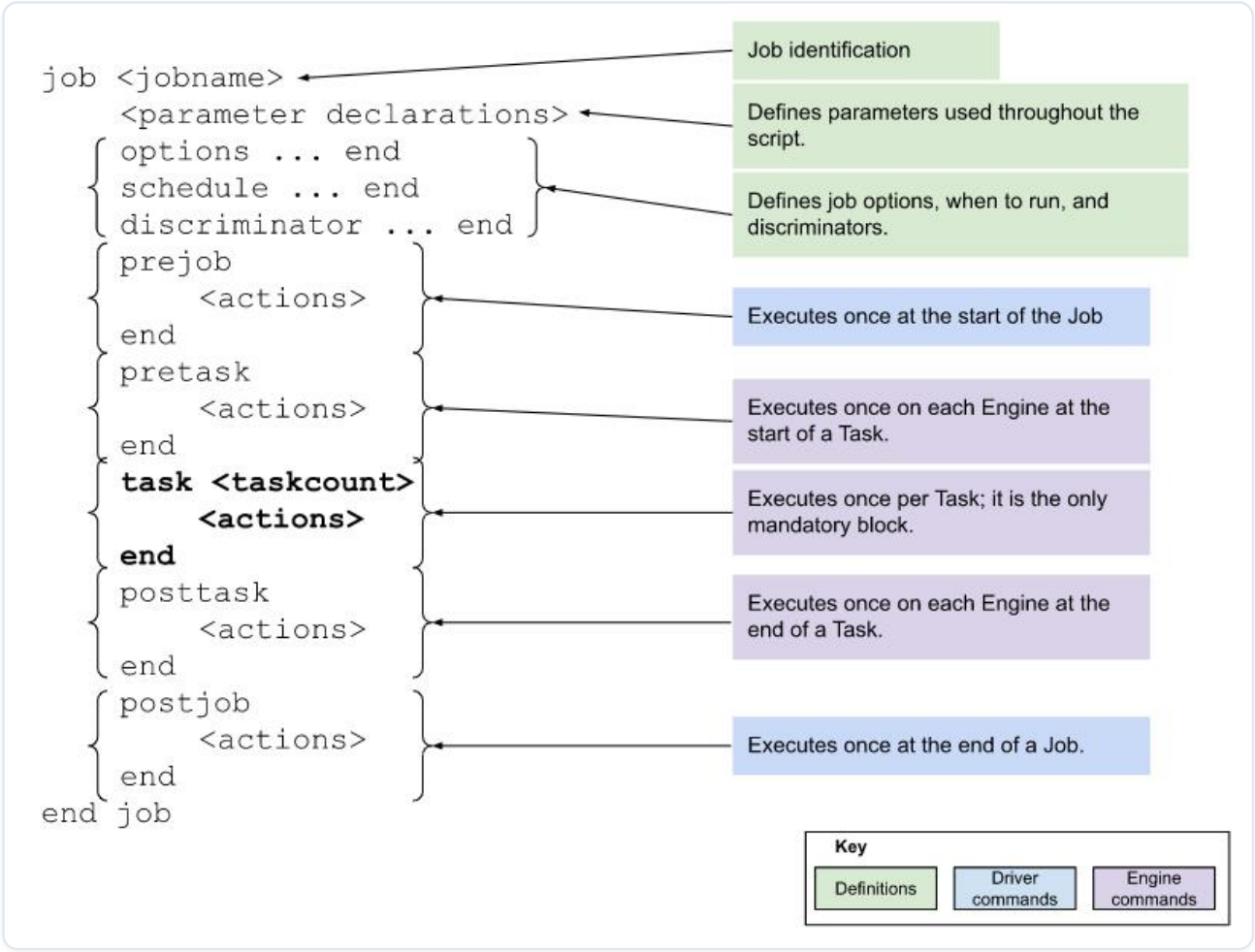
The PDS language contains constructs for carrying out various statements, such as running executables and copying files, at each point in the lifetime of a Service.

A few words about the lexical structure of PDS: whitespace is not significant, but the case is. All text from the “#” character to the end of the line is a comment and is ignored.

PDS Structure

A PDS file begins with the keyword `job` (a synonym for Service) and ends with the keyword `end`. Supply a name after `job` to identify the Service. Two types of elements can occur in between `job` and `end`: parameter declarations, which assign values to variables, and *blocks*,* which describe features or statements of the Service. The `options`, `schedule`, and `discriminator` blocks describe various facets of the Service, such as when to run it and which Engines can accept its tasks. The other five blocks describe statements to be executed at different

phases of the Service. All blocks are optional except for the `task` block. Multiple `job` blocks are also defined in a single PDS file, and they run sequentially by default. You can also run the `pdriver` command with the `-parallel` flag to run multiple jobs in one PDS file in parallel.



Structure of a PDS file

Details of the structure of a PDS file:

```
job jobname

options

    onerror ( fail | retry | ignore )

    maxFailedTaskRetries = val (default: 3)

    enableBlacklisting = true|false

    jobPriority = <val> (default: 5)

    autoCancelMode = "always" | "never" | "libloadfailure"
```

```
    jobOption "key" "val"
    jobDescription "key" "val"
end
<variable assignment>
schedule
    properties
    [email = "address"]
end
discriminator
    [ affects
        properties
      end ]
    properties
end
prejob
    statements
end
pretask
    statements
end
task taskcount
    statements
```

```
end

posttask

    statements

end

postjob

    statements

end

end
```

The Depends Statement

Multiple `job` blocks can be included in a single translation unit (a PDS file and any included PDS files), and by default, they run sequentially. It is also possible to define jobs that run based on the completion or failure of other jobs using the `depends` statement within a job block. For example, the job containing the following code runs if `firstjob` succeeds, if `secondjob` fails, and following `thirdjob` in either case:

```
depends

    firstjob succeeds

    secondjob fails

    thirdjob succeeds or fails

end
```

The Include Statement

You can include another PDS script within a PDS script by using the `include` statement. For example:

```
include "filename"
```

This includes a PDS script contained within `filename`. The filename can be declared with any string type. The filename is relative to the working directory from which you run PDriver, and can contain a relative or absolute path to a PDS file. The PDS file must contain a full job, and you can use it only outside the top-level `job` block. For example, a single PDS file can contain three include statements, each one including a `job` block stored in another file.

Lifecycle Blocks

Lifecycle blocks include statements that execute during the life of a Service. The following sections describe the lifecycle blocks.

Other blocks include the options block, discriminator block, and schedule block.

prejob

The prejob lifecycle block executes once at the very beginning of the job before any tasks are submitted to the Manager. These commands execute on the Driver.

pretask

The pretask lifecycle block executes once on every Engine that processes tasks for the job before any tasks are processed. Use this block for generic initialization such as obtaining input files common to all tasks.

task

Commands in the task lifecycle block execute once per task. The number of tasks in the Service is determined by the expression following the `task` keyword. Array variables referenced in the task block are indexed by the task ID, as explained in more detail in the Arrays section, below.

Since the same Engine can take more than one task, the statements in the `task` block run many times on the same Engine. The statements in the `pretask` and `posttask` blocks run only once per Engine.

The task block is the only required block in a PDS file. Thus a computation that only required executing a program ten times could be expressed by the PDS program:

```
job simple

    task 10

        execute "myprog"
```

```
    end

end
```

posttask

The posttask lifecycle block executes once on every Engine at some point after the job finishes. Place Engine cleanup tasks here. Note that the Engine's file cleaner automatically cleans all files in the `DSWORKDIR` directory. The posttask block typically executes after the postjob block executes on the Driver side, although it is not guaranteed to execute then. Execution order of the blocks in a PDS file is typically `prejob`, `pretask`, `task`, `postjob`, and `posttask`, but this is not guaranteed.

postjob

The postjob lifecycle block executes once on the Driver after all tasks finish. Use this block to obtain outputs from the staging directory, run post-processing scripts, and so on.

The Options Block

Use the options block to set Service options and description information.

Use the following directives for features specific to PDriver jobs:

Options Block Keywords and Arguments

KEYWORD	ARGUMENT	DESCRIPTION
`onerror`	`ignore`, `retry` or `fail`	The default way to handle errors. See "Builtin Commands" below for details.
`maxFailedTaskRetries`	A numeric expression	The number of times to resubmit a failed task.
`autoCancelMode`	`never`, `always` or `libloadfailure`	Whether to cancel the entire job when a single task fails. The default is `libloadfailure`, which cancels the job if a task fails due to the inability to load a library on the Engine side. You can set this standard

KEYWORD	ARGUMENT	DESCRIPTION
		option with `jobOption`, but the `autoCancelMode` directive enables you to use mnemonic strings, rather than numbers, as arguments.
`enableBlacklisting`	An expression evaluating to `true` or `false`	This argument is the same as `engineBlacklisting` for other Services. The default value is `false`.

Set standard options (described in the C++ API reference documentation for the `Options` object) with the `jobOption` directive:

```
jobOption "engineBlacklisting" "true"

jobOption "priority" "8"
```

Set elements of the job description with the `jobDescription` directive:

```
    jobDescription "serviceName" "Distributed Grep"
```

Each argument to `jobOption` or `jobDescription` must be a string or an expression that evaluates to a string. Literal numbers are not allowed.

The following `jobOption` directives are common to both PDriver and CPPDriver:

jobOption Directive Keywords and Arguments

KEYWORD	ARGUMENT	DESCRIPTION
`email`	string	An email address that is notified when a Service is completed.
`priority`	integer	The priority of this Service. The default value is 5.

KEYWORD	ARGUMENT	DESCRIPTION
`taskMaxTime`	integer	If a running task exceeds this amount of time in seconds, the task is rescheduled or retried based on the setting of `rescheduleOnTimeout`. The task is rescheduled or retried when the rescheduler checks for expired tasks after each poll period. This poll period is 60 seconds by default; it can be changed at **Admin > System Admin > Manager Configuration > Services**, under the **Service Rescheduler**heading, with the **Poll Period**property. The default value of `taskMaxTime` is infinite.
`autoCancel`	`0`, `1`, or `2`	Whether to automatically cancel the Service on a task failure. Possible values include 0 (`AUTO_CANCEL_NEVER)`, 1 (`AUTO_CANCEL_LIBRARY_LOAD)`, or 2 (`AUTO_CANCEL_ALWAYS)`. The default value is 1 (`AUTO_CANCEL_LIBRARY_LOAD)`.
`compressData`	`true` or `false`	Whether to compress the task, input, and output data. Compression time is minimal and recommended for data sizes greater than 10K for each input or output. The default is `false`.

KEYWORD	ARGUMENT	DESCRIPTION
`killCancelledTasks`	`true` or `false`	Whether to kill and restart an Engine if a task is canceled. If `false`, the canceled method is called rather than killing the Engine, to provide user-defined interruption of the task and any necessary cleanup. The default value is `true`. Tasks are canceled when canceled in the Administration Tool, when a Service is canceled, and when another Engine completes the task due to redundant rescheduling.
`tasksPerMessage`	integer	The maximum number of tasks for each submission or retrieval message. Regardless of this number, messages do not exceed 100 KB. The default is 100.
`autoPackNum`	integer	The number of tasks in an auto-packed task. In this mode, a tasklet processes multiple task inputs in one Service routine by packing task inputs into a single task and calling your Service routine on all inputs. Use this mode when there are more inputs than Engines, or tasks are of short duration, to maximize efficient use of memory and Engine processing power. If inputs are added outside of `createTaskInputs`, the Service checks every second if there are any unsubmitted

KEYWORD	ARGUMENT	DESCRIPTION
		tasks and submits them in a package even if there are fewer than the number requested, to ensure that all tasks are submitted. Task IDs in the Administration Tool are the IDs of the task packages, so they do not directly correspond to the task ID from the Driver and Engine's point of view. `TaskletException` auto-resubmission and task-level discrimination are not supported for this mode, and are ignored. Autopack is also not supported on multiplexed Engines. The default value is 0.
`sharedUnixDir`	string	A directory in which the Driver and UNIX Engines exchange data. This directory must be an NFS mounted directory to which all UNIX Engines working on this job have read/write access. This overrides use of the file servers on the Driver and Engines, and is optimally a directory local to this Driver for minimum network bandwidth. If set and using Windows Engines, the Windows shared directory must also be set to the equivalent of this directory.
`sharedWinDir`	string	A directory in which the Driver and Engines exchange data. This directory must be a

KEYWORD	ARGUMENT	DESCRIPTION
		Windows shared directory to which all Windows Engines working on this job have read/write access. This overrides use of the file servers on the Driver and Engines, and is optimally a directory local to this Driver for minimum network bandwidth. Typically, the share is Windows UNC format, such as `\\server\data`. If set and using UNIX Engines, the UNIX shared directory must also be set to the equivalent of this directory.
`checkpoint`	`true` or `false`	Enables checkpointing for this Service. The default value is `false`.
`maxEngines`	integer	The maximum number of Engines that can be working on a task at a time. The default value is infinite.
`statusExpires`	`true` or `false`	Whether the status of the job in the **Services > Services > Service Session Admin**page expires. If `false`, the status must be manually removed. The default value is `true`.
`engineBlacklisting`	`true` or `false`	Whether to prevent Engines that fail at a task from taking other tasks from that Service. The default value is `false`.

KEYWORD	ARGUMENT	DESCRIPTION
`unloadNativeLibrary`	`true` or `false`	Whether to unload the native library once the Service finishes. Set the value to `false` for sharing global objects in the library. The default value is `true`.
`deleteInvocationData`	`1` or `2`	When to purge Service invocation data from display in the Administration Tool. The default value is 1 (`SERVICE_COMPLETED),` to purge when a Service completes. Set to 2 (`SERVICE_REMOVED),` to purge when a Service is removed from the Administration Tool.
`maxTaskRetries`	integer	The maximum number of retries allowed for any task that fails. A retry occurs if the task failed and `serviceFailRetry` is true, or if the task exceed the `taskMaxTime` and `maxTaskReschedules` is false. The default value is 3.
`maxTaskReschedules`	integer	The maximum number of redundant reschedules allowed for any task, if any of the rescheduler strategies are in effect on the Broker. The default value is 3.
`rescheduleOnTimeout`	`true` or `false`	How a task is dealt with if it exceeds the `taskMaxTime`. If `true`, the request is rescheduled, and the current one

KEYWORD	ARGUMENT	DESCRIPTION
		continues. If `false`, the Engine running the task is killed, and the task is retried. The default is `false`.
`serviceFailRestart`	`true` or `false`	Whether an Engine restarts itself on a Service invocation failure. The default is `false`.
`serviceFailRetry`	`true` or `false`	Whether a Service request is retried on a failure. If `true`, it is retried up to the maximum number of times, as set by `maxTaskRetries`. The default is `false`.
`gridLibrary`	string	A Grid Library that is used for this Service. The string argument specifies the name of the Grid Library.
`gridLibraryVersion`	string	The version of the Grid Library that is used for this Service. The string argument specifies the version of the Grid Library.

JobDescriptions

All Services have a `JobDescription` object created upon instantiation, with default settings. Predefined properties are stored in the database. You can define any other properties. The following `JobDescription` s properties are set by default:

JobDescription Properties

PROPERTY	DESCRIPTION
`appName`	The application name.

PROPERTY	DESCRIPTION
`appDescription`	The application description.
`deptName`	A department name associated with the Service.
`groupName`	A group name associated with the Service.
`individualName`	An individual's name associated with the Service.
`serviceName`	The name of the Service. By default, this is the Service ID.
`class`	The name of the class in the Service.
`serviceType`	The type of Service.

The Discriminator Block

This block specifies either a job-level or task-level discriminator for the job. A discriminator without the `affects` clause is considered a job-level discriminator. You can declare only one job-level discriminator. The `affects` clause specifies conditions that must be met for the discriminator to be applied to a particular task. You can use multiple discriminator blocks with `affects` clauses.

Discriminator declarations consist of a property name, a comparator, and a value:

```
param-name ==|!=|<|>|<=|>= param-value
```

The property name refers to Engine properties. A predefined set of properties is assigned to all Engines by default. The administrator can assign additional properties to Engines at Engine Properties and Engine Property List. To see an Engine's properties, select **Engines > Engine Admin**, and from the **Actions**list, click **Engine Details**. Property names are case sensitive in PDS scripts. For example, the following discriminates against Engines with a value of less than 350 in the `cpuMHz` Engine property:

```
cpuMHz > 350
```

This example shows the format of the affects clause:

```
discriminator

    affects

        $DSTASKID        #variable

        <                #comparator

        10               #numeric or string

end
```

The variable can be any array type or built-in variable. `$DSTASKID` is the number of the current task, starting with zero. The comparator and numeric or string match against the literal to determine if the discriminator applies against this task. The example above applies a discriminator to the first ten tasks in a job.

The Schedule Block

Parameters in this block have an effect only if the job is submitted asynchronously with `bsub`. You can use these schedule declarations:

relative

```
type = relative

minuteDelay = val
```

absolute

```
type = absolute

startTime = "mm/dd/yy hh:mm AM|PM"
```

With either type, declaring `email="string"` in the Schedule block sends an email to the address given in the string when the job is complete.

Variables, Types, and Expressions

Basics

PDS has two primitive types, string, and floating-point number, with the usual notations. You need not declare variables. Variables can take on values of different types over time. To dereference a variable, precede its name with a dollar sign. Values are converted to the appropriate type depending on context. For instance, a string is converted to a number when it appears in an arithmetic expression. The grammar forbids certain combinations of expressions to catch common mistakes. Some examples:

```
a = 5.2

b = "2.5"

c = $a + $b            # succeeds, value is 7.7

d = 5.2 + "2.5"        # disallowed by the grammar
```

Variables, Types, and Expressions

Basics

PDS has two primitive types, string, and floating-point number, with the usual notations. You need not declare variables. Variables can take on values of different types over time. To dereference a variable, precede its name with a dollar sign. Values are converted to the appropriate type depending on context. For instance, a string is converted to a number when it appears in an arithmetic expression. The grammar forbids certain combinations of expressions to catch common mistakes. Some examples:

```
a = 5.2

b = "2.5"

c = $a + $b            # succeeds, value is 7.7

d = 5.2 + "2.5"        # disallowed by the grammar
```

Scoping

Variables assigned outside of any block are global and visible to all blocks. A variable assigned within a block is visible only within that block.

Inside a block, you can assign a variable as a global variable, which is visible within other related blocks. Use the following syntax:

```
global a = 5.2
```

The following table describes what blocks are related with respect to scope:

GLOBAL VARIABLE SET IN	EFFECTIVE IN
outside of the lifecycle blocks	all blocks
the `prejob` block	all blocks
the `pretask` block	pretask, task and posttask
the `task` block	task and posttask
the `posttask` and `postjob` blocks	that block

Note that assigning a variable with the same name as a previously defined global variable does not change the value of the global variable. Instead, it creates a new local variable in the block, and that local variable has local scope and does not change the value of the global variable.

Variable Substitution

Variable references are expanded within quoted strings in all contexts. For example, after

```
a = "foo"

b = "$a fighters"
```

the value of `b` is `"foo fighters"`. Use curly braces to separate a variable name from the adjacent non-whitespace text:

```
b = "${a}bar"            # b contains "foobar"
```

Inside quotation marks, represent a quote by escaping it with another quotation mark. Also, inside quotation marks, represent the dollar sign character by escaping it with another dollar sign. For example:

```
a = """hello"""            # a contains "hello"

b = "$$100"            # b contains $100
```

The backquote can also be used to assign the output of a shell command to a variable. For example:

```
datetime = ‘date‘
```

Expressions

PDS supports the usual arithmetic operators (`+`, `-`, `*`, `/`) with standard precedence. All arithmetic operations use double-precision floating-point values.

PDS also supports the standard C-style comparison operators `(==`, `!=`, `>`, `<`, `>=`, `<=)`. These operators perform numeric comparison if both arguments are valid numbers; otherwise, they perform string comparison. They evaluate to zero if the comparison is false and to a non-zero value if it is true.

PDS does not support the relational operators `and`, `or`, and `not`.

Backquote expressions

A string enclosed in backquotes (`‘such as this‘`) has variable substitution performed on it, and the result is evaluated in a subshell. The standard output of the command is collected, newlines and linefeed characters are replaced by spaces, and trailing whitespace is removed. The result is the value of the expression.

Arrays

Arrays are fundamental to achieving parametric parallelism in PDS, because an array variable is implicitly indexed in the task block.

Construction

You can construct arrays of primitive values in several ways. Write a literal value as follows:

```
a = [1, 2, 3, 4, 5]
```

You can construct an array with autorange expressions. The expression

```
begin n end m step k
```

constructs an array starting with *n* and proceeding in increments of *k* until *m* is reached. More precisely, it constructs

```
[n, n+k, n+2k, ..., n+rk]
```

where *r* is the largest integer such that *n*+*rk*<= *m*.

The expression

```
begin n count c step k
```

constructs an array of *c* elements beginning with *n* and proceeding in increments of *k*, that is,

```
[n, n+k, n+2k, ..., n+(c-1)k].
```

For example,

```
begin 10 end 15 step 2
```

and

```
begin 10 count 3 step 2
```

both construct the array

```
[10, 12, 14].
```

The third way to construct an array is to use the `split` function, which divides a string into array elements at whitespace. Quoted elements keep embedded whitespace and strip the quotes upon placement into the array. For example, on UNIX machines:

```
split(‘ls‘)
```

returns an array of the files in the current directory.

Indexing

In most contexts, when a variable containing an array is referenced, the first element is returned. However, in the task block, the element corresponding to the ID of the currently running task is returned. (If the task ID exceeds the array size, the last element of the array is returned.) This feature makes it easy to write the most common kinds of distributed computations. For example, you can set up an array of values and run a command on each one in parallel with the following PDS script:

```
args = begin 100 end 200 step 5

task sizeof($args)

    execute "doit $args"

end
```

The `$args` in the execute statement expands to 100 for the first task, 105 for the second, and so on.

Another exception to the first element being returned from an array is that inside a `for` loop, the variable containing the array returns the value at the current iteration of the loop. This is redundant with the loop variable. For example, in `for i in $args log "$i" log "$arg" end`, `$i` and `$arg` are the same every time; the loop variable `$i` is there only for convenience.

Explicit array indexing is unsupported: To obtain an array element other than the first, use one of the following:

- Implicit indexing in the task block
- The `for` statement

Even within those contexts, assignment to the variable holding the array changes the entire variable value, not the current element.

Other Features

An array that includes both string and numeric values is legal. Arrays of arrays are not allowed.

You can use the `sizeof` function to determine the number of elements in an array, as shown in the argument to the `task` block in the above example.

You can use the `for` statement to iterate over the elements of an array. See The For and Foreach Statement, for more information.

Built-in Variables

In addition to user-defined parameters, you can use the following built-in variables:

Built-in Variables

VARIABLE	DESCRIPTION
`DSTASKID`	ID of the current task (meaningless if not in task block)
`DSJOBID`	ID of the current job. If submitted using the `bsub` switch, this indicates the batch ID.
`DSWORKDIR`	The temporary directory for job files. This is an Engine-side variable. This directory is in the Engine installation directory, typically `./work/` *`machinename`* `-` *`i`* `/tmp/` *`session-ID`*, where machinename is the name of the Engine host; *`i`* is the instance, starting with 0; and *session-ID* is the Service Session ID for the PDriver Job. The Engine periodically deletes the `tmp` directory. To set the frequency of deletion, set **Temp File Time-to-Live (hours)**in each Engine Configuration.
`DSENGINEHOME`	The Engine home. This is an Engine-side variable.
`DSSTAGEDIR`	The Alias for the Manager staging directory. You can use this as a source or destination directory for the `copy` command. Files in this directory are automatically deleted periodically.
`DSOS`	The OS of the current system (for example, “linux”, “win32”, “plinux”)

It is important to put any temporary files in the `DSWORKDIR` directory. If you put them elsewhere and a task is interrupted, the files stay on the computer and are never automatically removed.

You can use automatic variables such as `$1`, `$2`, `$3`, and so on to reference arguments to the PDriver command that follow the script file name. You can use the automatic variable `$*` to reference these automatic variables collectively as a string. To create an array of all command-line arguments, use `split($*)`.

Arguments to the PDriver command that follow the script file name be referenced with the automatic variables `$1, $2, $3 ...` These variables also be referenced collectively as a string using the automatic variable `$*`. To create an array of all command-line arguments, use `split($*)`.

`$?` contains the execution status of the last `execute` command.

Statements

Built-in Commands

You can use the following built-in commands inside any lifecycle block (with the exception of `onerror` and `throw`). These commands are operating-system independent.

Built-in Commands

COMMAND	DESCRIPTION
`mkdir "dir-name"`	Creates a directory.
`copy "src-file" "dest-file"`	Copies a file. Typically, the prejob block is used to copy input files from the Driver machine to the staging directory on the Manager, which can be referenced as `$DSSTAGEDIR`. In the task or pretask block, the input file is copied from the staging directory to the Engine's local filesystem. Output files are copied in the reverse direction.
`rmdir "dir-name"`	Removes a directory. The directory must be empty.
`delete "file"`	Deletes a file. Supports the `*` wildcard.
`log "log-message"`	Writes a message to the Driver log (in prejob or postjob lifecycle blocks) or the Engine log (in pretask, task, or posttask lifecycle blocks).

COMMAND	DESCRIPTION
`execute [stdin="stdin=file"] [stdout="stdout-file"] [stderr="stderr-file"] "command-to-execute"`	Executes a command on the local machine using the shell specified with the `shell` command. A subshell is spawned for the command unless specified otherwise with the `shell` command. The subshell has the path available on the Engine (as set in the **Library Path**property of the Engine Configuration) or Driver. Before running custom executables, deploy those executables to Engines using GridServer's application resource deployment feature. See Resource Deployment for more details.
`shell "shell-type"`	Specifies the shell to spawn for commands run by the `execute` command. The defaults are `/bin/bash` for Linux and `cmd.exe` for Windows. If you set this to `none`, no shell is spawned.
`env (variable)`	Returns the value of the specified environment variable. If the variable is a string literal, enclose it in quotes. Returns an empty string for nonexistent variables.
`onerror <ignore|retry|fail>`	Indicates what happens when an `execute` command returns with a nonzero exit code. `ignore` takes no action; `retry` reschedules the task on another Engine; and `fail` returns an exception back to the Driver. The default is `fail`. When used with `retry`, you can use `onerror` only within the `job` or `task` blocks.
`throw "message"`	Causes the task to fail and throws an exception. The message is displayed on the Driver and written into the Driver log. This cannot be used in the posttask lifecycle block.

Enclose arguments to all statements in quotes, except for arguments to `onerror`. If you include a linebreak in a quoted argument, escape it with a backslash.

The `mkdir`, `copy`, `delete`, and `rmdir` commands are not dependent on the operating system. Pathnames automatically translate to work on the appropriate platform. For example, `mkdir "sample/log1"` creates `sample\log1` on Windows systems.

Statements

Built-in Commands

You can use the following built-in commands inside any lifecycle block (with the exception of `onerror` and `throw`). These commands are operating-system independent.

Built-in Commands

COMMAND	DESCRIPTION
`mkdir "dir-name"`	Creates a directory.
`copy "src-file" "dest-file"`	Copies a file. Typically, the prejob block is used to copy input files from the Driver machine to the staging directory on the Manager, which can be referenced as `$DSSTAGEDIR`. In the task or pretask block, the input file is copied from the staging directory to the Engine's local filesystem. Output files are copied in the reverse direction.
`rmdir "dir-name"`	Removes a directory. The directory must be empty.
`delete "file"`	Deletes a file. Supports the `*` wildcard.
`log "log-message"`	Writes a message to the Driver log (in prejob or postjob lifecycle blocks) or the Engine log (in pretask, task, or posttask lifecycle blocks).
`execute [stdin="stdin=file"] [stdout="stdout-file"] [stderr="stderr-file"] "command-to-execute"`	Executes a command on the local machine using the shell specified with the `shell` command. A subshell is spawned for the command unless specified otherwise with the `shell` command. The subshell has the path available on the Engine (as set in the **Library Path**property of the Engine Configuration) or Driver. Before running custom executables,

COMMAND	DESCRIPTION
	deploy those executables to Engines using GridServer's application resource deployment feature. See Resource Deployment for more details.
`shell "shell-type"`	Specifies the shell to spawn for commands run by the `execute` command. The defaults are `/bin/bash` for Linux and `cmd.exe` for Windows. If you set this to `none`, no shell is spawned.
`env (variable)`	Returns the value of the specified environment variable. If the variable is a string literal, enclose it in quotes. Returns an empty string for nonexistent variables.
`onerror <ignore|retry|fail>`	Indicates what happens when an `execute` command returns with a nonzero exit code. `ignore` takes no action; `retry` reschedules the task on another Engine; and `fail` returns an exception back to the Driver. The default is `fail`. When used with `retry`, you can use `onerror` only within the `job` or `task` blocks.
`throw "message"`	Causes the task to fail and throws an exception. The message is displayed on the Driver and written into the Driver log. This cannot be used in the posttask lifecycle block.

Enclose arguments to all statements in quotes, except for arguments to `onerror`. If you include a linebreak in a quoted argument, escape it with a backslash.

The `mkdir`, `copy`, `delete`, and `rmdir` commands are not dependent on the operating system. Pathnames automatically translate to work on the appropriate platform. For example, `mkdir "sample/log1"` creates `sample\log1` on Windows systems.

The If Statement

The syntax of the if statement is

```
if expression comparisonOperator expression then

    statements

end
```

PDS also supports `elsif` and `else` clauses. You can use the `if` statement inside a lifecycle block (`prejob`, `pretask`, `task`, `posttask`, `postjob`) to conditionally execute statements. One typical use is to execute different commands depending on the Engine's operating system:

```
if $DSOS == "win32" then

    cmd = "dir"

else

    cmd = "ls"

end

execute "$cmd"
```

An `if` statement can also appear at the top level of a PDS script, to include or exclude global assignments or entire blocks. An example is

```
if $1 == "alwaysCancel" then

    options

        autoCancelMode "always"

    end

end
```

The `if` statement is not legal in the `options`, `schedule`, or `discriminator` blocks.

The For and Foreach Statement

Explicit looping over an array is not often needed. If you need explicit looping, use the `for` statement:

```
for variable in expression

    statements

end
```

The expression must evaluate to an array. The variable takes on successive elements of the array on each iteration of the loop. Assignment to the loop variable is legal but has an effect only for the remainder of that iteration. It does not alter the array.

Inside a `for` loop, the variable containing the array returns the value at the current iteration of the loop. This is redundant with the loop variable. For example, in

```
for i in $args  log "$i" log "$args" end
```

the `$i` and `$args` are both equal for each iteration. The loop variable `$i` is there only for convenience.

The `foreach` statement lets you implicitly index an array, without the loop variable:

```
foreach expression

    statements

end
```

Shell Directives in Heterogeneous Environments

It is the PDS script writer's responsibility to declare a shell directive that is appropriate to the executing node's platform. For jobs in heterogeneous environments, you can specify different directives within an if-then-else block. For example,

```
if $DSOS == "win32" then shell "cmd.exe" else shell "/bin/bash" end
```

You can specify the argument “none”, in which case all following `execute` tasks spawn directly and not in a subshell.

It is also possible to use the bash shell with Windows by utilizing Cygwin, the UNIX environment for Windows.

To use bash and Cygwin on Windows machines:

1. Install Cygwin on another machine. You can get it at `www.cygwin.com`.
2. From the Cygwin installation, copy `bash.exe` and `cygwin1.dll` into the Engine’s path. This is either in a directory within the `%PATH%` on the Engine or the replication directory `<engine_home>/resources/win32/bin`. Also, copy any other Cygwin executables you use from the bash shell.
3. If bash is already in the path for the Engine, use the following syntax in the PDS script:

```
task

    shell "bash.exe"

    ...

end
```

or

```
task

    shell "none"

    execute stdout="outfile"

          "<optional_path>/bash.exe -c ""mycommand.exe"""

end
```

If the path to bash is not set, use the following syntax:

```
task

    shell "c:\cygwin\bin\bash"
```

```
    execute stdout="outfile" "echo hello"

end
```

The above example also requires that you copy the Cygwin `echo.exe` to the Engine.

Example

`pi.pds` performs a distributed Monte Carlo calculation of the value of pi. A `picalc` command-line executable resides in the `bin/`*platform* directory of the distribution. The executable is also available on all Engines in your GridServer installation. Running PDriver distributes calculation of Pi across the network, using autoranged parameters to generate differing seed values for the program's pseudorandom number generator. The postjob block runs a script that derives an average of all the values returned.

Creating Grid Libraries

This section provides information about creating Grid Libraries, which are versioned sets of resources that might be used by multiple Services.

Overview

A *Grid Library* provides a solution to the problem of managing versioned sets of resources that might be used by multiple Services. A Grid Library is a set of resources and properties necessary to run a Grid Service, along with configuration information that describes to the GridServer environment how those resources are to be used. For example, a Grid Library can contain JARs, native libraries, configuration files, environment variables, hooks, and other resources.

Grid Libraries are identified by **name**and **version**. All Grid Libraries must have a name and typically have a version. The version is used to detect conflicts between a required library and a library that has already been loaded; it also provides for automatic selection of the latest version of a library. A GridServer Service can specify that it is implemented by a particular Grid Library by specifying the `gridLibrary` and `gridLibraryVersion` Service Options or Service Type Registry Options.

Grid Libraries can specify that they depend on other Grid Libraries; like the Service Option, such dependencies can be specified by the name, and optionally the version. Also, nearly all aspects of a Grid Library can be specified to be valid only for a specific operating system. This means that the

same Grid Library can specify distinct paths and properties for Windows and Linux, but only the appropriate set of package options is applied at run-time.

Grid Library Format

The Grid Library can be an archive file in ZIP (`.zip`) or gzipped TAR format (`.tgz` or `.tar.gz`), with a `grid-library.xml` file in the root. It might also contain any number of directories that contain resources. Although the filename has no inherent meaning, the following format is strongly encouraged:

```
library_name-library_version.[zip|tar.gz|tgz]
```

If a Grid Library has the `OS` attribute set, the following format must be used:

```
library_name-library_version-os.[zip|tar.gz|tgz]
```

To reduce Engine restarts when you deploy a new version, always include the version in the filename, as overwriting existing libraries requires a restart, whereas a new library does not.

The directory structure is completely up to you since the configuration file is used to specify where resources are found within the Grid Library.

The configuration file must be a well-formed XML file named `grid-library.xml`, and be in the root of the Grid Library.

The GridServer SDKs include a `grid-library.dtd` file that can be used to validate the XML file. They also include an example Apache Ant `build.xml` file that can be used to validate and build Grid Libraries. This DTD can also be found in The grid-library.dtd.

Following is a table that specifies all elements and attributes of the `grid-library.dtd` file. It uses the XML schema notation for elements and attributes, such as:

(Required)

? (Optional)

* (Optional and Repeatable)

Grid Library DTD Elements and Attributes

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
`grid-library`	The root element. Note that attributes affect deployment; Engines only download Grid Libraries whose attributes match their own properties and ignore those with non-matching attributes. If no attributes are specified in this element for a particular Grid Library, all Engines download this Grid Library.	ELEMENTS • `grid-library-name` • `grid-library-version?` • `arguments?` • `dependency*` • `conflict*` • `jar-path*` • `lib-path*` • `assembly-path*` • `command-path*` • `hooks-path*` • `environment-variables*` • `java-system-properties*`
		ATTRIBUTES • `jre` • `bridge?` • `super?` • `os?` • `compiler?`
`grid-library-name`	The library name. All libraries must be named.	

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
`grid-library-version`	The version. If not specified, 0 is implied. If in comparable format as defined below, it can be used to determine the latest version.	
`dependency`	A library dependency. If the version is not specified, the latest version is chosen at runtime.	ELEMENTS • `grid-library-name*` • `grid-library-version?` ELEMENTS • `os?` • `compiler?`
`conflict`	Indicates that this library conflicts with the given library. If this Grid Library is NOT a dependency, and `grid-library-name="*"`, then it indicates that this Grid Library conflicts with all other Grid Libraries (aside from its dependencies).	ELEMENTS • `grid-library-name*`
`pathelement`	An element containing a relative path, typically set to a directory. This element must be in the proper format for the OS.	
`jar-path`	The JAR path. If specified, all JARs and classes in the path are loaded.	ELEMENTS

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
		• `pathelement*` ATTRIBUTES • `os?` • `compiler?`
`lib-path`	The native library search path.	ELEMENTS • `pathelement*` ATTRIBUTES • `os?` • `compiler?`
`assembly-path`	The .NET assembly search path. Absolute assembly paths, mapped drives, and UNC paths do not work.	ELEMENTS • `pathelement*` • `os?` • `compiler?`
`command-path`	The path in which the Engine searches for Command Service executables.	ELEMENTS • `pathelement*` ATTRIBUTES • `os?` • `compiler?`
`hooks-path`	Path to Engine Hook XML files. Engine Hooks are initialized at the time the containing Grid Library is loaded.	ELEMENTS • `pathelement*` ATTRIBUTES • `os?` • `compiler?`

ELEMENT	DESCRIPTION	ELEMENTS AND ATTRIBUTES
`name`	The name of a property	
`value`	The value of a property	
`property`	A name/value pair, used by environment variables and Java System properties.	ELEMENTS • `name, value`
`environment-variables`	Environment variables to set.	ELEMENTS • `property` ATTRIBUTES • `os?` • `compiler?`
`java-system-properties`	Java system properties, which are set immediately prior to executing a task using this library.	ELEMENTS • `property` ATTRIBUTES • `os?` • `compiler?`

The following is a list of attributes used above. Valid values can be found in the **About**page in the Administration Tool:

Grid Library Attributes

ATTRIBUTE	DESCRIPTION
`os`	The `os` attribute specifies that it is only applied to this OS. If the attribute is not this operating system (OS), the containing element and its children and content are ignored.

ATTRIBUTE	DESCRIPTION
`compiler`	If the attribute is not this compiler, the containing element and its children and content are ignored.
`super`	If set to `true`, the Grid Library is a Super Grid Library, meaning, it is always loaded when the Engine starts up. See Super Grid Libraries for more information.

Variable Substitution

You can use placeholder variables in a `grid-library.xml` file, which are then substituted with their value as defined in a properties file or in an OS environment variable. This enables quick changes in properties in the `grid-library.xml` file without redeploying the Grid Library.

If the `grid-library.xml` file contains a property with a value contained with the `$` character, such as `$mydir$`, it is substituted with the value in one of three places, in this order:

- A default properties file in your Grid Library named `grid-library.properties`. This can provide baseline values for your variables.
- An external properties file, named with the same name as the Grid Library archive, with the extension `.properties`, in the Grid Library deployment directory. Values in an external properties file replace those defined in the default properties file within the Grid Library.
- A defined OS environment variable. This value replaces the value defined in either properties file.

> **Note:**
>
> If the substitution is not found in the file, the empty string, "", is substituted.

> Substitutions are allowed anywhere in a string within the content of property value elements and pathelements. Multiple substitutions per string are allowed. `$` characters can be treated as literals by escaping them with another `$` character. Windows paths that are specified in the `[library].properties` file must escape the `\` character with another `\`.

The implicit property `ds.GridLibraryRoot` can be used in the `grid-library.xml` file. This is useful for including a `driver.properties` file in a Grid Library, and setting the `DSDRIVER_DIR` to the location of the file in the library. This can be done in a C++ or .NET Grid Library as follows:

```
<environment-variables>

<property >

<name>DSDRIVER_DIR</name>

<value>$ds.GridLibraryRoot$</value>

</property>

</environment-variables>
```

The following would be done in a Java Grid Library:

```
<jar-path>

    <pathelement>$ds.GridLibraryRoot$</pathelement>

</jar-path>
```

Versioning

Versioning provides the following functionality:

- It enables deployment of new versions of libraries and deletion of old versions without interrupting currently executing Service Sessions.
- It provides for specifying conflicts, or libraries that cannot coexist with each other.
- It enables a Service Session or dependency to specify the use of the latest version of a Grid Library.
- It enables easy management of multiple applications needing different versions of the same set of libraries.

To use versioning, you must specify the Grid Library version in the configuration file. An Engine can load only one version of the library with the same name at any time. If the version is not specified, it is implied to be 0.

The version must be a String that follows the proper comparable version format. It can also be used to determine the latest version of the library, for automatic loading. This format is

```
[n1].[n2].[n3]...
```

where `nx` is an integer, and there might be one or more version points.

For instance,

```
4.0.1.1, 4.1, 3
```

are in the proper comparable version format.

The integer at each version point is evaluated starting at the first point, and continues until a version point is greater than the other. If a version point does not exist for one, it is implied as zero.

For instance

```
4.0.0.1 > 4.0

4.0.0.5 < 4.0.1.1
```

To specify that a dependency or Service use a particular version of a Grid Library, the version field is set to that value. To specify that it uses the latest version, the field is left blank.

If a version is specified it must match exactly. That is, `3.0.0` is not the same as `3`; if the library's version is `3.0.0` and the Service specifies `3`, the Service does not find that library and subsequently fails.

If a version is specified but not in this format, and there are multiple versions of a library, the "latest version" is undefined. Thus, automatic selection of the latest version is only possible when all Grid Libraries with the specified name provide a version in the proper format.

By default, if a Service was set to use the latest version of a Grid Library, all Engines work on the latest version at the time the Service was started, regardless of whether a newer library has been deployed. This can be changed by setting the `GRID_LIBRARY_STRICT_VERSIONING` Driver option to false. When false, if a newer version of the library is deployed while the Service is running, Engines that have not yet worked on the Service use the newer version, while Engines that worked on it prior to deployment continue to use the older version.

Dependencies

Grid Libraries might specify dependencies on other Grid Libraries. A dependency specification resolves to a particular Grid Library using two values:

- **grid-library-name** The name of the Grid Library, as specified in the dependency's XML
- **grid-library-version** The version of the Grid Library, as specified in the dependency's XML. OS compatibility is determined by checking the `os` and `compiler` tags for the top-level element in the dependent Grid Library. If not specified, it uses the latest version supported by the OS.

Note that if a dependency resolves to more than one Grid Library, the dependency used is undefined.

Two dependent libraries conflict if they have the same library name, but different versions.

It is possible to specify an OS attribute to `<dependency>` element for ignoring Grid Libraries that do not apply to an Engine's particular operating system. For example, if a Grid Library contains native libraries for multiple platforms, you can specify OS-specific dependencies on the bridge Grid Libraries such that the Engine only loads the bridge corresponding to its operating system.

Note that if a dependency is missing, the Engine logs a warning. Rather than the current task failing, the Engine attempts to continue loading all necessary libraries to run the task.

Conflicts

A conflict between two Grid Libraries means that these libraries cannot be loaded concurrently. When there is a conflict between a loaded Grid Library and a Grid Library required by a Service, the Engine must restart to unload the current libraries and load the requested library.

The following circumstances result in a conflict:

- **Version Conflict** The most common conflict arises via versioning, and typically when upgrading versions or using more than one version of the same library concurrently. This conflict arises when a Grid Library with the same `grid-library-name` as the requested Grid Library, but a different version, is loaded.
- **Explicit Conflict** There can be situations in which different Grid Libraries can conflict with each other due to conflicting native libraries, different versions of Java classes, and so on. Because the Engine cannot determine these implicitly, the `conflict` element can be used to specify Grid Libraries that are known to conflict with this Grid Library.

Additionally, the value of the grid-library-name can be set to `"*"`. This means that this Grid Library can conflict with all other Grid Libraries (aside from its dependencies), and it is guaranteed that no other Grid Libraries load concurrently with this Grid Library. Note that this is only allowed if the Grid Library is not a dependency; if the `"*"` is used as a conflict in a Grid Library that is a dependency, a verification error occurs.

- **Dynamic Version Conflict** A Grid Library conflict occurs if dynamic versioning is used, and the latest version of a Grid Library or Grid Library dependency has changed due to an addition or removal of a dependency since the Grid Library has been loaded.
- **Variable Substitution Conflict** A Grid Library conflict occurs if its variable substitution file has changed since it has been loaded.

Grid Library Loading

When a Service Session is set to use a Grid Library, that library is loaded. Loading is the process of setting up all resources in the Grid Library for use by the Service. A library is loaded only once per Engine session.

First, the library loads itself, and then it loads all dependencies. The library loader uses the depth-first, or preorder traversal algorithm when loading libraries. When there are a number of dependencies in a Grid Library, the order in the XML is considered left-to-right with respect to the algorithm. The library search order for `lib-path` and `jar-path` is determined by their respective lists. Certain aspects of a load might require a restart, and possibly re-initialization of the state. The following steps are performed by a load of the root library and all dependencies:

1. Checks for conflicts with currently loaded Grid Libraries. If so, it restarts with the requested Grid Library and clears out the current state of any loaded libraries.
2. If new `lib-path` s have been added for its OS, they append to the current list of `lib-path` s. The state of loaded libraries includes all libraries already loaded, plus the requested library. Note that specifying a JRE dependency has this effect.
3. If new `jar-path` s have been added for its OS, the jars and classes are added to the classloader.
4. If new `assembly-path` s have been added, it adds them to the .NET search path.
5. If new `command-path` s have been added for its OS, it is added to the search path for Command tasks.
6. If new `hooks-path` s have been added, any hooks in the path are initialized.
7. If the default is current and a Grid Library is requested, the Engine restarts.

State Preservation

Under most cases, when an Engine shuts down, it preserves the current state of which Grid Libraries it has loaded. When it starts back up, it loads all Grid Libraries that were loaded when it shut down. As Grid Libraries are loaded, the pathelements they contain are added to a ‘master’ list of paths for that type of pathelement. For example, if a Grid Library contains a `lib-path` specification, that `lib-path` is appended to the list of `lib-path` values obtained from already-loaded Grid Libraries.

Note that this means that it is up to the creator of the Grid Libraries deployed on the grid to ensure that the ordering of library paths does not lead to loading the wrong library. For example, if two different Grid Libraries each provide DLLs in their `lib-path` s that share the same name, because of OS-specific library load conventions, the one used is the first one in the aggregate `lib-path` from across all loaded Grid Libraries. Likewise for Java classes, when more than one copy of the same class is in the classloader, it is undefined which class loads. Therefore it is important to either subdivide Grid Libraries appropriately when such conflicts arise or to use the `conflict` element to explicitly state conflicts.

Grid Library and RunAs State information persists on normal Engine shutdowns, which includes task failures aside from crashes. If the Engine does not shut down normally, such as if it crashes, or if the Daemon kills the process due to it exceeding the shutdown timeout, the state is reset.

If an Engine shuts down due to a conflict, it clears the current state and sets up for only the requested Grid Library upon restart. This is referred to as preloading. If an Engine shuts down due to internal library inconsistencies or a crash, the state is not saved. State is also cleared on all instances when a Daemon is disabled and reenabled.

Task Reservation

If an Engine requires a restart to load a Grid Library, the task is reserved on the Broker for that Engine. The Engine is instructed to log back into the same Broker, and takes that task upon login. The timeout for this is configurable at **Admin > System Admin > Manager Configuration > Services**.

Environment Variables and System Properties

All Environment variables and Java System properties for a Grid Library and all dependencies are set each time a task is taken from a particular Service that specified that Grid Library. (They are not cleared after the task is finished.) Environment variables are set via JNI so that they can be used by native libraries or .NET assemblies, and they are also passed into Command Services. Note that environment variables such as `PATH` and `LD_LIBRARY_PATH` must not be changed through

this mechanism. Rather, `library-path` and `command-path` are reserved for manipulating these variables.

Using Grid Libraries from a Service

Services can specify a Grid Library to use by setting the `GRID_LIBRARY` and optionally the `GRID_LIBRARY_VERSION` Service Options. This would typically be set by Service Type in the **Service Types**page, although it can be set programmatically on the Session. Services can specify a Grid Library to use by setting the corresponding Service Option values. If the version is not set, a Service uses the latest version of a Grid Library.

If a Service needs to find resources in a Grid Library, it can use the Grid Library Path. This value is a path value that includes the root directories of all Grid Libraries currently loaded. For Java, .NET, and C++, the path is `EngineProperties.GRID_LIBRARY_DIR;` for command Services, it is the environment variable `ds_GridLibraryPath`.

Super Grid Libraries

A Grid Library can be declared as a Super Grid Library. This means that it is always loaded when the Engine starts up. The typical use case for this is to have an `EngineHook` that queries the system for some information, which is used to set `EngineSession` properties prior to the Engine running any tasks.

To specify that a Grid Library is a Super Grid Library, set the `super` attribute in the `grid-library` element. For example, `<grid-library ... super="true" />` . Super Grid Libraries also cannot have conflicts or dependencies. Other libraries cannot depend on or conflict with them.

Super Grid Libraries are loaded upon startup before anything else. They are ignored on conflict checks for `*` (all).

If a new Super Grid Library is deployed while an Engine is running, it is loaded. If a new version of an existing Super Grid Library is deployed while an Engine is running, the Engine restarts.

C++ Bridges

C++ Bridges are the native bridges that allow Engines to execute native Services. They are packaged as Grid Libraries, named `cppbridge-[os]-[compiler]-[M]-[m]`, where `M` and `m` are the GridServer major and minor version numbers. All C++ Bridges are pre-packaged and deployed in the Grid Library replication directory upon GridServer Manager installation or upgrade.

JREs

In the rare event that a particular service cannot use the default JRE that is deployed to the Engines, a JRE can be packaged as a Grid Library. The Service's top-level Grid Library would then declare it as a dependency. When an Engine takes a Task, it then restarts using this JRE. Note that the JRE must be a supported version.

JREs are packaged as `jre_name-version.gz` or `jre_name-version.zip` where, `jre_name` includes `jre-os`. The `version` is the JRE version, for example, `1.8.0.331`. The os is the platform, such as `linux64`, `win64`, `linux`, or `win32`.

For example, for linux 64: `jre-linux64-1.8.0.331.tar.gz`.

To package a JRE Grid Library, create a Grid Library that contains the JRE in a directory in the root of the Grid Library. Then, set the `jre` attribute in the `grid-library` element. For example, `<grid-library ... jre="true" />`

For a JRE Grid Library, you can optionally specify JVM arguments in the XML. To do so, add an `<arguments>` element to the root element. It can take any number of `<property>` elements, each containing a `<name>` element and an optional `<value>` element.

If the property has a value, the argument `name=value` is added. Otherwise, just the `name` argument is added.

If the same argument is set in the Engine Configuration and the Grid Library, the Grid Library overrides the Engine Configuration.

> **Note:**

> Specifying the JVM debug port inside a Grid Library results in unpredictable behavior and is not supported. Set this functionality with the **Debug Start Port**setting on the **Grid Components > Engines > Engine Configurations**page.

> For an example of how to package a JRE Grid Library, see the "Deploying Services" section of the *TIBCO GridServer® Administration*.

R Grid Libraries

Grid libraries with Services implemented in R must contain a top-level directory named `R`. This means the following Grid Library structure is used:

```
/

/grid-library.xml

/R/
```

The `R` directory is required unless the Grid Library contains only R packages. All files ending in `.R` are sourced. The order that the R files are sourced is not guaranteed.

Grid libraries with R Services must depend on the `rbridge-`*platform* Grid Library. That dependency can be direct or indirect.

Grid Libraries can also contain R packages in non-source form. Packages root directory within the Grid Library is defined using the `lib-path` element.

The following is an example `grid-library.xml` with R packages:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library os="win64">

    <grid-library-name>Rpscl-win64</grid-library-name>

    <grid-library-version>3.0.3</grid-library-version>

    <lib-path>

        <pathelement>library</pathelement>

    </lib-path>

    <dependency os="win64">

        <grid-library-name>rbridge-win64</grid-library-name>

    </dependency>

    <dependency os="win64">

        <grid-library-name>Rextras-win64</grid-library-name>

    </dependency>
```

```
</grid-library>
```

Building TERR Runtime Grid Libraries

A tool to create a TERR runtime Grid Library is provided in the SDK's `$SDK_HOME/tools/bin` directory.

The `createTERRGL.bat` and `createTERRGL.sh` scripts are a wrapper for the Java implementation. Therefore, Windows and Linux versions take the same parameters:

OPTION	DESCRIPTION
`--terrHome=${TERR_HOME}`	Specify the `TERR_HOME` directory. Required.
`--version=${gl version}`	Override the version information deduced from `${TERR_HOME}/bin/TERR --version`.
`--32|--64`	Generate 32-bit or 64-bit Grid Library. By default, Windows generates a 32-bit runtime, and Linux generates a 64-bit runtime.
`--complete`	Generate Grid Library with all files. By default, smaller Grid Libraries are generated without some unused files.
`--no-build-info`	Generate Grid Library without build information. By default, build information is in the Grid Library descriptor.
`--verbose`	Generate more output.
`--help`	Show a list of all options.

Python Bridges

Python Bridges are the native bridges that enable Engines to execute Python scripts using the packaged Python runtime. They are packaged as Grid Libraries, named `pybridge-`*os*`-`*compiler*`-`*M*`.`*m*, where *M* and *m* are the GridServer major and minor version numbers. All Python Bridges are pre-packaged and deployed in the Grid Library replication directory upon GridServer Manager installation or upgrade.

Python Grid Libraries

Grid Libraries with Services implemented in Python must contain a top-level folder where the individual `.py` files are stored. In the example below, this folder is named `commands`. In the `grid-library.xml`, you must reference this folder in a `lib-path` element.

The Grid Library must either contain the binary ("pybridge-win64-vc14" for Windows or "pybridge-linux64-gcc34" for Linux) or reference a dependent grid-library containing it.

Lastly, it is necessary to set an environment variable `ds.PYTHON_USER_FILE_PATH` which references the folder containing the `.py` files set above. The value of this environment variable must be `$ds.GridLibraryRoot$/`*folder-name* where *folder-name* is the folder containing the `.py` files.

Here is an example of a `grid-library.xml` for Python:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library>

    <grid-library-name>Python-example</grid-library-name>

    <grid-library-version>1.0.0</grid-library-version>

       <lib-path>

        <pathelement>commands</pathelement>

    </lib-path>

    <dependency os="win64">

      <grid-library-name>pybridge-win64-vc14</grid-library-name>

      <grid-library-version>7.1</grid-library-version>

   </dependency>

   <dependency os="linux64">

      <grid-library-name>pybridge-linux64-gcc34</grid-library-name>

      <grid-library-version>7.1</grid-library-version>
```

```
    </dependency>

    <environment-variables>

     <property>

       <name>ds.PYTHON_USER_FILE_PATH</name>

       <value>$ds.GridLibraryRoot$/commands</value>

       </property>

    </environment-variables>



</grid-library>
```

Windows Application Deployment

The Windows Deployment Scripting Language provides a mechanism by which programs can be executed in conjunction with file updating on Windows Engines. This can be used for such purposes as registering COM DLLs and .NET assemblies, running Microsoft Installer packages, and so on. It runs an installation command when the script is added, and when any dependent files are modified. It can also run an uninstallation command when the script is removed.

A deployment script is a file named `dsinstall.conf`. This is a reserved filename, and the Engine Daemon interprets any file with this name as a deployment script. The script is a properties file, with name and value pairs that govern the command execution.

The script is placed, with associated files, in its own subdirectory of a Grid Library. This is referred to as the installation directory. In situations where a remote application installation must take place when the Engine is first started, it can be packaged in a Super Grid Library.

The following properties are provided:

Deployment Script Properties

PROPERTY	DESCRIPTION
`install_cmd`	The installation command. The command must be in the current directory; you can also specify the full path to a command. This command is run when the `dsinstall.conf` file is added, modified, and when any dependency is modified. Note that the Engine's `PATH` environment variable is not set within the installation command. You must set the path to use some commands, such as `ping` or `find`.
`workdir`	Working directory from which the commands are launched. The directory is relative to the installation directory.
`uninstall_cmd`	Optional. The uninstall command. This is executed when the script is deleted, or prior to subsequent runs of the install command if `uninstall_first` is true. Supporting files for the uninstall script might be deleted along with the script; the command is executed prior to local deletion of the files. Typically an uninstall is performed by simply removing the entire installation directory. Note that the path is not set within the uninstall command. You must set the path to use some commands, such as `ping` or `find`.
`waittime`	Number of seconds to wait for the install/ uninstall command to finish. The default is 30 seconds. If this time is exceeded, the process running the command is killed.
`uninstall_first`	Optional. If true, the uninstall command always runs prior to the install command, except for the first time the install command is run. This is for situations in which you need to uninstall software prior to reinstallation.

PROPERTY	DESCRIPTION
`success_exit_codes`	Optional. Comma-delimited list of exit code values that indicate successful command execution. If the exit code does not match any value, an error logs with the failure code, and the next time the Daemon restarts it retries the installation. If this property is not set, exit codes are ignored.
`disable_on_fail`	If an Engine Daemon disables itself on the failure of an install. The default is false if not specified in the conf file. When the value is true, the Engine Daemon disables itself if the installation returned exit code is not in the success exit codes.

The `:` and `\` characters must be escaped with a backslash (`\`) character in the `dsinstall.conf` file. Also, you must not rename the `dsinstall.conf` file.

The following code is an example of a script that installs a Microsoft Installer package:

```
workdir=.

waittime=30

uninstall_first=true

install_cmd=install.bat

uninstall_cmd=uninstall.bat

success_exit_codes=0

install.bat:

%SystemRoot%\system32\msiexec /q /i mypackage.msi ALLUSERS=1

uninstall.bat:

%SystemRoot%\system32\msiexec /q /x mypackage.msi ALLUSERS=1
```

These three files, plus the `mypackage.msi` file, are all placed in a Grid Library. Note that the `uninstall_first` property is used to uninstall the previous version of the software whenever the package is changed. To uninstall the software, simply undeploy the Grid Library; the uninstallation is performed prior to deleting the files.

Grid Library Example

The following example `grid-library.xml` is for a mixed Java/C++ application that runs on Windows, and both `gcc` and `gcc34` for Linux:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library>

    <grid-library-name>MyLib</grid-library-name>

    <grid-library-version>1.0.0.1</grid-library-version>

       <!-- Example of how to use both gcc and gcc34 libraries -->

    <lib-path os="linux" compiler="gcc">

        <pathelement>lib/gcc</pathelement>

    </lib-path>

    <lib-path os="linux" compiler="gcc34">

        <pathelement>lib/gcc34</pathelement>

    </lib-path>

<!-- All three C++ bridges are included here -->

    <dependency>

        <grid-library-name>cppbridge-vc14</grid-library-name>

    </dependency>

    <dependency>

        <grid-library-name>cppbridge-gcc</grid-library-name>
```

```
    </dependency>
    <dependency>
        <grid-library-name>cppbridge-gcc34</grid-library-name>
    </dependency>
<!-- Specifies that win32 use this JRE Grid Library, others use default --
>
    <dependency>
        <grid-library-name>jre-win32</grid-library-name>
        <grid-library-version>1.8.0.161</grid-library-version>
    </dependency>
<!-- Specifies JVM options in a JRE Grid Library -->
    <arguments>
        <property>
            <name>-Xdebug</name>
        </property>
        <property>
            <name>-Xmx512m</name>
        </property>
        <property>
            <name>-Dfoo</name>
            <value>bar</value>
        </property>
```

```
    </arguments>
<!-- Example of linking to another of my Grid Libraries-->
    <dependency>
        <grid-library-name>MyCalculator</grid-library-name>
    </dependency>
    <hooks-path>
        <pathelement>hooks</pathelement>
    </hooks-path>
<!-- Example of multiple jar paths -->
    <jar-path>
        <pathelement>jars</pathelement>
        <pathelement>morejars</pathelement>
    </jar-path>
<!-- Example of a lib path with absolute dir -->
    <lib-path os="win32">
        <pathelement>lib\win</pathelement>
    </lib-path>
<!-- Example of OS-dependent env vars, using a property sub -->
    <environment-variables os="win32">
        <property >
            <name>MY_WIN_VAR</name>
            <value>$WinVar$</value>
```

```
        </property>
    </environment-variables>
    <environment-variables os="linux" compiler="gcc">
    <property >
            <name>MY_GCC_VAR</name>
            <value>$LinuxDriverDir$</value>
        </property>
    </environment-variables>
    <java-system-properties>
        <property>
            <name>foo</name>
            <value>bar</value>
        </property>
    </java-system-properties>
</grid-library>
```

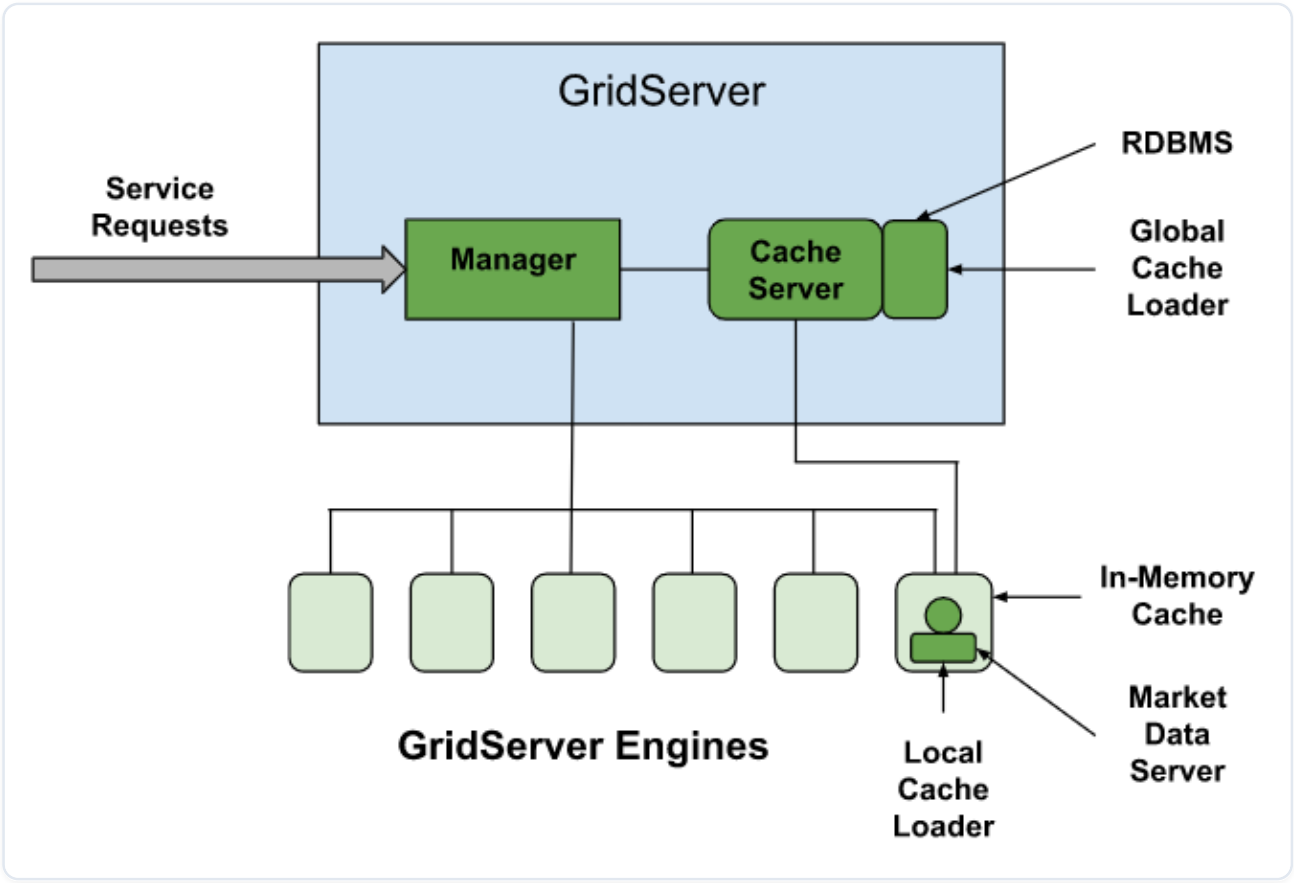
GridCache

This section provides information about using GridCache, a dynamically updateable distributed object cache that any GridServer Driver or Engine can use to store data for later retrieval by other GridServer components.

Overview

GridCache is a dynamically updateable distributed object cache that any GridServer Driver or Engine can use to store data for later retrieval by other GridServer components. While GridServer extensively uses object caches internally, applications can access the GridCache object cache

directly through an API, to reduce the load on backend datastores and decrease access time to data. GridCache is similar to the proposed JSR-107 JCache, with a similar interface and a subset of features.



A typical GridCache implementation

GridCache meets the requirements of many informational market data systems, where a consistent view of object state is extremely important but it is not necessary to guarantee that all participants process every individual state change (for instance, every individual quote move) as a transaction.

General Capabilities

GridCache is a general distributed cache that provides a consistent view of data to all clients (Drivers and Engines) in the Grid. Data resides in unique *regions* of the cache. Data can be serializable Java objects, .NET objects, strings, and byte arrays for C++. The global cache of data can be arbitrarily large, limited only by the amount of disk space on the Manager. Each component locally caches only the data requested by users on that component. The local cache of each client, designed to speed up access to frequently used data, is in-memory with the option in Java Drivers and Engines to spool to disk. The size of the local cache is configurable through the Engine distribution configuration or the Driver properties file.

API

You can access GridCache through a client API available on Drivers and Engines. The API follows the JCache specification where appropriate. The API is available in Java, .NET, and C++, and provides cross-platform access to data where appropriate. That is, C++ and Java applications can share XML documents (strings), but have little use for sharing Java or .NET objects. You can also use Data References across different platforms to support streaming very large objects. See Data References for more information.

Modes

GridCache operates in one of the following modes:

- **Local** This mode lets you cache data locally by putting elements into the cache. This mode does not synchronize clients that are accessing local cache regions with the same name. This is similar to having a local hashtable with LRU and eviction based on time-to-live from creation time.
- **Local with loader** This mode lets you load data into the local cache using a loader specified at create time. Puts (cache writes) are not allowed in this mode. Users can manually synchronize clients' local caches using clear and invalidate methods.
- **Global** Data that users put into the cache is then available globally. Full automatic synchronization occurs in this mode. All components have access to a synchronized view of all entries.
- **Global with loader** Users can use a global loader to load data into the cache. Full automatic synchronization occurs in this mode.

Cache Configuration and Access

To configure cache regions, use the GridServer Administration Tool at **Services > Services > Cache Regions**. Define the region names or regular expressions with a set of attributes. The `getRegion` method in `CacheFactory` provides access to the region if it already exists or creates the region with the mapped attributes. A region name throws an exception when it matches multiple regular expressions from different schemas. A second Cache access method is available and takes the schema name to provide a dynamic mapping of regions to attribute schemas.

When you configure Local and Global loaders, the class name of the loader and the type are arguments to the constructor. Users can define bean properties for loaders. Loaders are available only in Java and .NET, but can be used by the C++ API. JNI, or managed C++ can be used to

implement loaders to access native resources. Each schema that requires a loader defines the loader within the configuration page.

Changes to the cache configuration take effect the next time you create regions with that cache configuration. Pre-existing regions that require configuration changes must be manually destroyed and recreated.

Data Storage

All data put to a global cache is stored in a persistent backend datastore on the Broker's file system. You can limit the size of the Broker's file cache. The default is that there is no limit. If the Broker's file cache is size limited and full, the Broker silently removes data to make room for the incoming data. If the cache entry is too big, it first goes to the memory cache, and if it cannot fit there it goes to the disk cache. If it is too big for the disk cache too, then it is dropped. There is no global resilience when using a memory-based cache; however, you can use a loader for that purpose.

Attributes

Define the GridCache Configuration Attributes in schemas. Then, apply the attributes to newly created regions. You can define the following types of attributes:

- **TimeToLive**Regions can define a time-to-live attribute. Data that is stored in the cache for longer than the time-to-live attribute is evicted, and you must reload that data from the backend datastore. Data in local caches is evicted locally. Data for distributed regions is evicted from the distributed cache, and the Broker and all clients delete that data if they have cached it locally.
- **Local/GlobalLoader**Loads data from a backend datastore. See Cache Loaders, below.
- **KeepAlive**Specifies how long the client keeps the region and its keys in its local cache after the last reference to the region goes away.

Consistency/Synchronization

The cache synchronizes by propagating update notifications to all clients listening on a region. These notifications occur any time the region changes. Specifically, they occur on a put (when an element already exists), clear, remove, or invalidate. This applies to different region types differently:

- **Engines**GridCache guarantees that all Engines receive all update notifications by the time they take the next task or Service request.
- **Drivers**There are no synchronization guarantees for the Driver. The Driver receives notification messages the next time it performs a request, polls the server for results, or sends a heartbeat.

Cache Loaders

Loaders provide an optional mechanism for loading data into the cache from a backend datastore, such as a relational database. Users can implement and associate Cache Loaders with a region of the cache. These Cache Loaders can be installed locally on the client (Driver or Engine) or globally, in the GridServer Broker.

Global

Use a Global Cache Loader for synchronized regions from which all clients can access data.

- Global loaders are defined and configured in the schemas.
- When other clients get access to that region, they automatically are using that loader.
- A client can specifically pre-load data into the cache by explicitly calling the load method with a single key or a list of keys.
- If a get does not find data, the loader then attempts to load for that key by calling the loader's load method.
- Puts are not allowed on regions with loaders.
- Global loaders are written in Java, but can be bridged to native or .NET code through JNI.
- Global CacheLoader JAR files are deployed to the `lib` directory in the cache directory. By default, the cache directory is `DS_DATA/cache`. Configure the cache directory in the **Cache** section of the **Manager Configuration**page in the Administration Tool.

Local

Use a local loader to cache data locally from an external database. Clients do not share local loaders.

- Puts (writes) are not allowed, as it is a local cache, and data is not propagated to other clients or regions.
- Removing an item is not allowed. Instead, invalidate the item. Invalidating causes the item to be removed from other clients' caches.
- Local loaders can only be .NET or Java. You can adapt CPP loaders through JNI or managed C++.

Cache Loader Write-through and Bulk Operations

To simplify using backend datastores with GridCache, it's sometimes desirable to add a means for supporting the store and remove methods on the loader rather than just supporting it purely as a data fetching mechanism. It's also sometimes desirable to be able to use such methods in a batch

form, such that a cache can be “primed” with entries through a bulk load method, or that multiple objects can be stored, removed, and invalidated with a single method to cut down on per-store operation overhead.

Loaders can inherit from two interfaces that support methods for supporting the store and remove methods on the loader:

- The preload methods are exposed in the `BulkCacheLoader` interface.
- The store, remove, and clear methods are exposed in another interface, `CacheStore`, and can be used by the underlying cache mechanism for modifying the content of cache puts and removals on the backend datastore. The cache mechanism can then invalidate the corresponding entry or entries for all other caches listening on that region, if applicable.

It is possible that the backend store gets updated without sending an invalidation message to all other clients. If this scenario is detected, it throws an exception indicating a loss of synchronization, but the cache client must handle recovery from that point on.

The caching system in GridServer does not provide a mechanism to auto-update data in the cache when it changes in the backend, if done so by a mechanism other than those offered by the CacheStore interface.

Support is available for datastore write-through, bulk write-through, remove, bulk remove and bulk load, on both global and local loaders, in Java and .NET.

Notification

GridCache provides an optional mechanism whereby you can implement a class that listens for update notifications. An update is defined to be either an invalidation call on a loaded object or on a put call on a key that exists in the cache already. You can then take any action desired such as updating local copies of the object or data to the new version or ignoring the update completely. The next time that the data is requested from the cache, GridCache fetches and locally caches the most current version of that data.

Disk/Memory Caching

When the cache is full, cache puts (or writes) push the oldest element out of the cache. Elements are then put into the backing disk cache or removed entirely. This makes the caches LRU caches. You can configure the size of the local cache and the size of the backing disk cache:

- In the `driver.properties` file, for Drivers.
- In the Engine configuration, for Engines.

If you configure disk caching, then any puts into the memory cache when the memory cache is full force the oldest element out of the memory cache into the disk cache. Any access to a cache element that has to get the element from the disk cache brings the element into the memory cache. CPP Drivers do not have disk-backed cache.

Cache Region Scope

Global cache regions exist until they are destroyed through the destroy method regardless of whether any client has a reference to that region. Unnecessary global cache regions impact eviction performance, so it is important to destroy global regions when they are no longer needed.

After all references to a cache region on a client go out of scope, local cache regions persist on clients until their keepalive timeout. At that point, the region is swept from the cache. Use the `close()` method to explicitly release a reference to a region. If you do not use the close method, garbage collection handles decrementing references to the region. However, garbage collection is never guaranteed so the keepalive timeout is not a guaranteed timeout. Using the close method is recommended.

Data Conversion Matrix

The following table outlines the results of putting a data type into GridCache and then getting it with a different Driver.

INPUT	.NET OUTPUT	JAVA OUTPUT	C++ OUTPUT
String	String	String	std::string
Java or .NET byte	byte	byte	std::string
.NET Object	Object	undefined	undefined
Java Object	undefined	Object	undefined
DataReference	DataReference	DataReference	DataReference

Using The GridCache API

Details about the GridCache API are in the GridServer API JavaDoc, which is available from the Documentation page of the Administration tool. Documentation for the `Cache` interface covers the use of GridCache.

The GridCache API supports the following seven primitives:

GridCache constructor with CacheFactory

To create a new GridCache instance, use the CacheFactory to get a reference to a particular region. On a particular client component, you can construct multiple instances of a GridCache with the same region, but each exposed instance with the same region shares the same underlying implementation. This lets multiple Sessions share the same view of a cache without having to duplicate the storage or the code.

Put and Get

The `put` method writes to the cache a new entry for a key and object. The `get` method returns the object stored in the cache, for a given key. If you use the get method on a key that does not exist and the region has an associated loader, the loader attempts to load the data for that key. A second get method, when given a mapping of keys, bulk-gets a mapping of keys and their objects. Bulk get methods use a single HTTP request for each map, which can lower transaction overhead.

Region

Region names are any printable low ASCII character (32-126), except for characters prohibited in XML attribute values (', ", &, <).

Keys

A Key is a string that refers to an object in the cache. The `keys` method gets:

- For a global region type, a list of all keys currently stored on the Manager for this cache.
- For a local region type, a list of locally cached keys.

Remove

Removes this object from the region, from the Manager, and from all distributed caches.

Clear

Clears all objects from the region, the Manager, and regions on other components.

Invalidation handlers

By default, GridCache implements a lazy invalidation mechanism where callers are told only that their version of an object is out-of-date when they make a fresh “get” call for the object. The invalidation handler interface lets the caller register or deregister to receive asynchronous notification that a get, put, remove, or clear has invalidated the caller’s local copy of an object.

Note that .NET cache loaders for GridCache must be packaged as a Super Grid Library. For more information about using Super Grid Libraries, see the *TIBCO GridServer® Administration*.

Fault Tolerance and GridCache

GridCache supports fault-tolerance. For details, see the *TIBCO GridServer® Administration*.

GridServer Design Guidelines

This section discusses two important aspects to consider when designing an application to run on GridServer: data movement, and task or Service request duration. There are a variety of ways to move data among the machines involved in an application; the first section considers their characteristics and suggests which to choose under various circumstances. When you divide a problem into a set of tasks or Service requests, you can usually select how many to use, or equivalently, how much time each one can take. The second section discusses factors that can influence this decision.

Data Movement

Every distributed computation ultimately executes as a local computation—a single computing process. You must move every piece of input data across the network from wherever it resides to the machine that needs to process it, and every piece of output data must travel over the network from the machine that produced it to its ultimate destination. Additionally, you can use caching to optimize data movement, providing a strategy for lowering the amount of data transfer. Moving large amounts of data over a network efficiently is a crucial aspect in the design of most distributed applications. Efficient data movement can often make a dramatic difference in performance.

Principles of Data Movement

Good data movement design can be summarized in two principles:

- Move each piece of data over the network as few times as possible—preferably just once.

  The less that data is moved, the less time it takes to move it. But the many layers of abstraction offered by modern computer systems can hide data movement, making it harder to see the bottlenecks. Network file systems are a good example: there is no way to tell from reading the code whether a file is being read from the local disk or over a network, but the performance difference can be significant.
- Move data as early as possible—preferably before the computation starts.

Doing so improves the performance of the computation because the stopwatch that times the computation is started after the data movement has already occurred. But this is more than a mere accounting trick. Consider a nightly report that must run after 5 PM to avoid conflicting with daytime Services. If the data for the report is available at 4 PM, it can be distributed to Engines in the hour before the report runs.

Data Movement Mechanisms

The GridServer software uses the following data movement mechanisms:

- Service Request Argument and Return Value
- Service Session State
- Shared Directories and Direct Data Transfer
- Resource Update
- GridCache
- Data References

Service Request Argument and Return Value

The most direct way to transmit data between a Grid client and an Engine is through:

- The argument to a Service request and
- The return value from the Service request.

If you enable Direct Data Transfer, the data travels directly between Driver and Engine.

Each request is handled efficiently, but the aggregate data transfer across hundreds of requests adds up significantly. Therefore, factor data common to all requests into session state or init data, or distribute it by another mechanism.

Service Session State

Any Service Session can have an associated state. As described in the Services sections, this state resides on the Driver as well as on each Engine hosting the instance, so it is fault-tolerant with respect to Engine failure.

Service Session state is ideal for data that is specific to a session. Service Session state is easy to work with because it fits the standard object-oriented programming model; it is downloaded once per Engine.

Transmission of the Service Session state from Driver to Engine is peer to peer and is GridServer's Direct Data Transfer (DDT) feature. DDT is enabled by default. When DDT is enabled and a Service creation or Service request is initiated on a Driver, the initialization data or request argument resides on the Driver, sending only a URL (and not data) to the Manager. When an Engine receives the request, it downloads the data directly from the Driver rather than the Manager. This mechanism saves one network trip for the data and can result in significant performance improvements when the data is much larger than the URL that points to it, as is usually the case. It also greatly reduces the load on the Manager, improving Manager throughput and robustness.

Shared Directories and DDT

Some network configurations are more efficient using a shared directory for DDT rather than the internal file servers included in the Drivers and Engines. In this case, configure the Driver and Engines to read and write requests and results to the same shared network directory, rather than to transfer data over HTTP. All Engines and the Driver must have read and write permissions on this directory. Configure shared directories at the Service level with the `SHARED_UNIX_DIR` and `SHARED_WIN_DIR` options. If you use both Windows and UNIX Engines and Drivers, configure both options to be directories that resolve to the same directory location for the respective operating systems.

Resource Update

GridServer's Resource Update mechanism replicates Grid Libraries, or archives of versioned sets of resources, with Engines. It also replicates the contents of a directory on the Manager to a corresponding directory on each Engine. When you use Resource Update, use the **Services > Services > Grid Libraries**page in the GridServer Administration Tool to upload files to the Manager. After all currently running Services finish, the Engines download the new files. For more on Resource Update, see the TIBCO GridServer® Administration.

Resource Update is the best way to guarantee that the same file is on the disk of every Engine in your Grid. File Update is ideal for distributing application code, but it is also a good way to deliver configuration files or static data to Engines before your computation starts. Any kind of data that changes infrequently, like historical data, is a good candidate for distribution in this fashion.

GridCache

GridServer's GridCache feature is a repository on the Manager that is aggressively cached by components (Drivers and Engines). The repository comprises a set of regions, each of which is a map from string keys to arbitrary values. The GridCache API supports reads, writes, removing key-

value pairs, and getting a list of all keys in a catalog. For more information about GridCache, see GridCache.

A GridCache component caches every value that it gets or puts. If a component changes a key's value or removes it, the Manager asks all components to invalidate their cached copy of that key's value.

GridCache is fault-tolerant with respect to Engine failure because the data is stored on the Manager. When an Engine fails, its cached data is lost and its task is rescheduled. The Engine that picks up the rescheduled task gradually builds up its cache as it gets data from the Manager.

GridCache is a flexible and efficient way for Engines and Drivers to share data. Like File Update, an Engine needs only a single download to obtain a piece of constant data. Unlike File Update, GridCache supports data that changes over the life of a computation.

You can use GridCache for having Engines post results. This is generally only useful if those results are to be used as inputs to subsequent computations.

Data References

GridServer Data References are objects that represent data existing on a GridServer client. You can use them to pass lightweight data from one client to another so that only the destination needing the data performs the data transfer. Typically, one client filesystem stores the data and another client's file server serves the data.

Data Movement Examples

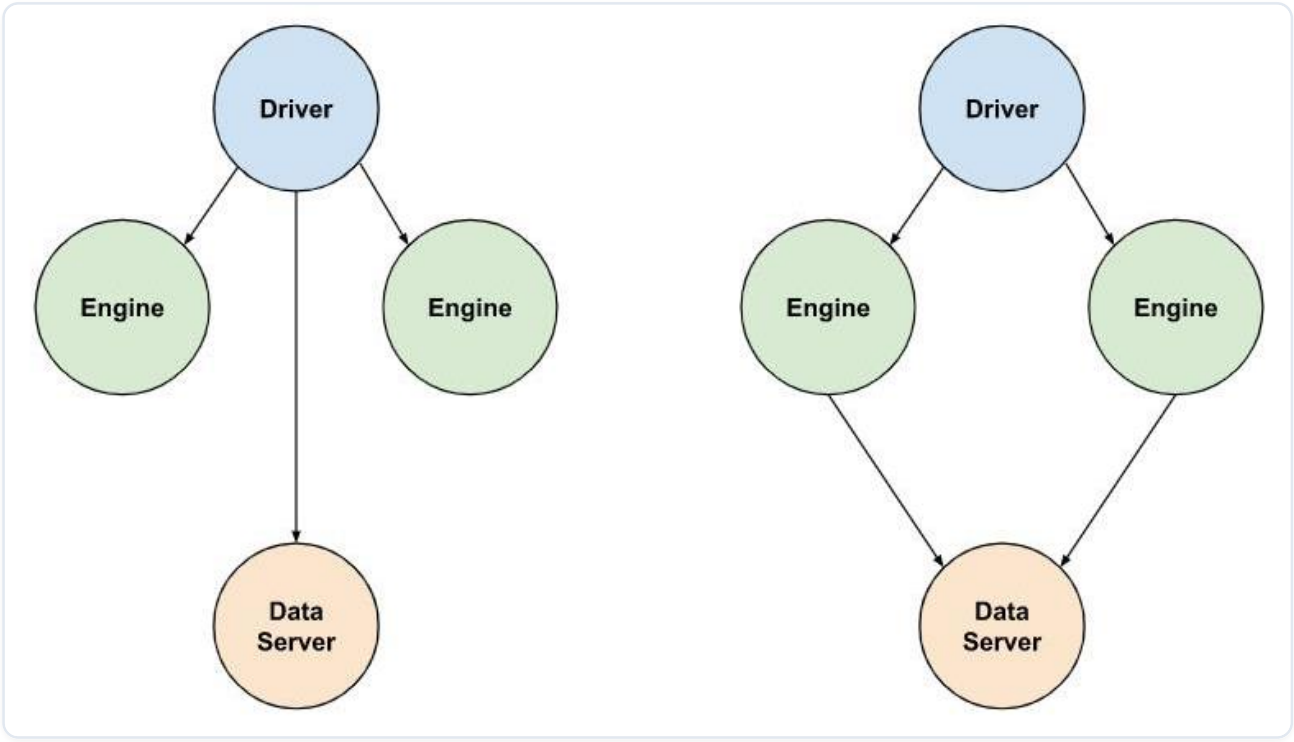
As an example of using the data movement mechanisms discussed above, consider the problem of determining the value of a financial instrument. This example uses a computation method named value. The value method takes two arguments: a deal and a pricing scenario. The deal argument contains all information specific to a financial instrument needed to determine its value, such as coupon and maturity date. The pricing scenario argument contains all other determinants of the deal's value, such as interest rates and prices of underlying instruments. The output of the value function is a single number representing the value of the deal under the given pricing scenario.

Typical applications require the value of many deals over one or several pricing scenarios. To distribute and parallelize this computation, we execute the value function simultaneously on many Engines. We assume the code for the value function is available to each Engine (whether by Resource Update or over a network file system). We also assume that the numbers returned by the value function make their way back to the client through the standard Service return value

mechanism. The question we want to consider is how to get the deal and pricing scenario information to the Engines.

Database Access

We first look at the deal information itself, stored in a database or data server somewhere on the network. Compare the two scenarios in Figure 8-1. In the first diagram, on the left, the Driver loads the deal information from the data server and sends it to the Engines. In the second diagram, on the right, the Driver sends just the unique identifier and has each Engine access the data server on its own.



Data flow between a Driver, two Engines, and a Data Server

The second choice is better because it requiresfewer data moves across the network to accomplish the same result. In the first choice, the data moves across the network twice, once from the data server to the Driver and second from the Driver to the Engine. In the second choice, the data moves across the network only once from the data server to the Engine. Also, the data needs marshaling and unmarshaling only once.

The second choice also increases parallelism at the data server. In the first choice, only the Driver is attempting to load data from the data server. In the second, multiple Engines attempt to load data concurrently. Assuming that the data server can handle the load, the second choice increases parallelism.

Single Pricing Scenario

We now consider the case in which you use a single pricing scenario to evaluate many deals. Here is one (suboptimal) way to organize this computation. We assume throughout that you already deployed and registered a Service containing the value function.

Algorithm 1 (suboptimal):

1. Create a Service Session of the value Service.
2. For each deal, submit the **deal identifier**and the **pricing scenario**as an asynchronous request to the Service Session.
3. Wait for results.

   Although this algorithm gets the job done, it needlessly sends the same pricing scenario multiple times.

   This is an ideal application of Service Session state:

Algorithm 2:

1. Create a Service Session of the value Service, initialized with the pricing scenario.
2. For each deal, submit the deal identifier as an asynchronous request to the Service Session.
3. Wait for results.

By making the pricing scenario be part of the session's state, it is transmitted only as many times as there are Engines that implement the session, rather than once per request. GridServer never allocates more Engines to a Service session than there are requests for that instance, so Algorithm 2 never moves more data than Algorithm 1. And in the likely event that there are many more requests than Engines (we argue below in the Task Duration section why this is a good idea), Algorithm 2 moves much less data than Algorithm 1.

Several Pricing Scenarios

What if the application needs to value the portfolio of deals for more than one pricing scenario? One approach is simply to repeat Algorithm 2 several times, creating a new Service session for each pricing scenario. It is also possible to use a single session and employ the `updateState` method of the Service client API to transmit each successive pricing scenario to the Engines running the session. If the differences between pricing scenarios are small and they are used to perform the update instead of the pricing scenarios themselves, then using `updateState` can result in considerable data movement savings; even if the pricing scenarios themselves are used as updates, this approach is still likely to be superior to using separate instances.

Multiple Pricing Scenarios Available Early

Now let us add the following wrinkle: we still want to compute the value of many deals over many pricing scenarios, but the pricing scenarios are available to us sometime before we can run the application. For instance, pricing scenario information is available at 4 PM, but we cannot start the nightly report until 5:30 PM, to avoid interfering with daily work. In this situation, we can exploit the time gap to push information to the Engines before the computation starts. One approach would be to use File Update to put all the pricing scenario data on all the Engines. Another would be to put the pricing scenario data into GridCache and run a “primer” Service that copies the data to the Engines. The trade-offs between these two approaches were discussed above under Data Movement Mechanisms.

Deal-Pricing Scenario Symmetry

Finally, we point out that deals and pricing scenarios are for the most part symmetric in these examples (the main difference being that pricing scenarios are less likely to be indexed by primary key in a database, so the discussion of deal identifiers versus deal data does not apply to them). For instance, if deals are available to you early, you can use File Update or GridCache to push deal information to Engines before your application starts.

Service or Task Duration

Service or Task duration has an important impact on the performance of distributed computations. Recall that a task corresponds to a single Service request when using Services. Make tasks long enough to compensate for communication overhead, but not so long that their interruption seriously delays the overall computation. Dividing the work into more tasks, each of which takes less time, can also mitigate the performance degradation that can arise from having tasks of different sizes. We discuss these issues in detail in the following sections.

As a running example, we use the deal valuation problem discussed in the previous section on data movement. There we assumed that each task was responsible for pricing a single deal. But this is unlikely to be efficient for most types of deals; instead, group several deals together in a single task.

Engine Interruption and Smoothing

If an Engine is interrupted or fails during a task, that task must run again from the beginning. Therefore, divide work into tasks with short execution times. The shorter the task, the less work you lose when an Engine fails.

Additionally, shorter tasks result in better performance. This is because you are reducing the variability of task durations in a computation.

For example, suppose that you divide the work of computation so that 10 Engines each have one task. You expect that this minimizes communication overhead and that Engines do not fail. However, what if you wrongly estimate one task and it takes twice as long as the others? Since all tasks must finish for the computation to be complete, the longest task determines the computation time. If you have nine one-minute tasks and one two-minute task on 10 Engines, the computation takes two minutes, with the last minute consisting of nine idle Engines and one Engine still working on the two-minute task. With exactly as many tasks as Engines, your program runs as long as the longest task. (This section simplifies this discussion by ignoring communication time.)

Suppose you use twice as many tasks as Engines. This significantly improves the expected running time. To understand why, continue the above example. If you divide each of the 10 tasks in two, you have 20 tasks for 10 Engines: 18 30-second tasks, and two one-minute tasks. Each Engine takes two tasks at random. The chance of the same Engine receiving both long tasks is fairly small, so this program is likely to take one and a half minutes most of the time.

Similarly, more, shorter tasks smooth out the effect of different processor speeds. Assume that all tasks take the same time, but that one Engine is slower than the others. With exactly one task per Engine, the slow Engine determines the computation time. With many short tasks, the slow Engine takes fewer tasks than the other Engines, and all Engines finish at close to the same time, minimizing the time for the whole computation.

Summary

Communication overhead dictates using long tasks, but the possibility of Engine failure and the opportunity to smooth over differences in task durations and processor speeds suggest using many quick tasks. The best compromise is to choose a task running time between 30 seconds and several minutes and to choose a number of tasks that is three or four times the number of available Engines.

These are just a few examples of how to improve the performance of your Services. For a more complete list, see the *Performance and Tuning* section of the *TIBCO GridServer® Administration*.

The Admin API

The GridServer Admin API offers programmatic access to administrative tasks or information normally performed or presented in the web-based GridServer Administration Tool. The API is available through Java, C++, and .NET Drivers and Services, on Managers through a Server Hook, from a SOAP Web Service, with JMX, and REST.

Documentation for the GridServer Admin API

Detailed documentation on GridServer Admin API is in the API documentation. The WSDL is available in the GridServer Administration Tool at **Grid Components > Drivers > Web Services**.

The following components are defined:

- `BatchAdmin`
- `BrokerAdmin`
- `DriverAdmin`
- `DriverManager`
- `EngineAdmin`
- `EngineDaemonAdmin`
- `ManagerAdmin`
- `ServiceAdmin`
- `UserAdmin`
- `Version`

The methods allowed are based on the Security Roles of the user. When called in a Service, the user is considered to be the user that created the Service Session. When called in a Server Hook, all methods are available. Note that methods return null if there is no output, as opposed to returning a zero-length array. For example, `EngineAdmin.getAllEngineInfo()` returns null if there are no Engines currently logged in to the Broker.

See the **Admin > User Admin > Role Admin**page on the Administration Tool for a list of all permissions.

Using the Admin API over SOAP

The following example uses the Admin API over SOAP with Java:

1. Locate the WSDL for the Service from the Manager's Web Service List. For example, for the `EngineDamonAdmin` class use, http://example:8080/livecluster/webservices/EngineDaemonAdmin?wsdl
2. Generate Java Stubs for the Service. For example, using Axis:

```
org.apache.axis.wsdl.WSDL2Java

http://example:8080/livecluster/webservices/EngineDaemonAdmin?wsdl
```

3. Use the Stubs. For example:

```
// Get the interface to the Admin Service

EngineDaemonAdmin server = (new
EngineDaemonAdminServiceLocator()).getEngineDaemonAdmin();

// Required when Driver authentication is enabled

((Stub)server).setUsername("admin");

((Stub)server).setPassword("admin");

// Maintain the session ID for each request

((Stub)server).setMaintainSession(true);

// Query the Admin Service

EngineDaemonInfo[] info = server.getAllEngineDaemonInfo();
```

Using Server Hooks

The entire Java Admin API is available within a Server Hook. For details about implementing Server Hooks, see the JavaDoc documentation for the `ServerHook` class.

Using JMX

The Java Admin API is also available using JMX. Most API components are exposed as MBeans within the `com.datasynapse.gridserver.admin` tree. See the JavaDoc documentation for the `com.datasynapse.gridserver.admin` package for more information about each object.

Using Conditions

In a typical Grid environment, machines are not all identical. Some machines are slower, or have less RAM; other machines are faster but work to capacity during the day. Depending on the Services you have and the general demographics of your computing environment, the scheduling

of Services to Engines might not be clearly deterministic. And sometimes, a specific Service might require special handling to ensure that optimal resources are available for it.

Conditions

Conditions are a feature of GridServer. Conditions affect how Service Sessions and tasks are scheduled to Engines. Conditions enable you to use Engines selectively based on their properties.

GridServer provides the following types of Conditions:

- **Discriminator Conditions**specify a subset of Engine that can work on a task or Session, typically based on Engine properties.
- **Affinity Conditions**enable you to set an affinity for tasks or Sessions for Engines, based on Service state and typically properties. Unlike Discriminators, they do not prevent tasks from going to any Engine, it only attempts the best match.
- **Dependency Conditions**enable submitting workflows to a Broker without an active Grid client to manage Dependencies (wait for completed tasks, submit more based on a successful outcome, and so on).
- **QueueJump Conditions**enable the Driver to specify that a Task is placed at the front of the waiting queue, rather than at the back.
- **Descriptor Conditions**set descriptive information about an Invocation request.
- **EXTRAConditions**prevent Engines from taking Tasks when a necessary resource, such a database connection, is not available. The current resource counts are updated with a simple REST API.

Discriminator Conditions

Use Discriminator Conditions to select Engines for particular Services based on Engine properties. Discriminators have many uses:

- Limit a Service to run only on Engines whose usernames come from a specified set, to confine the Service to machines under your jurisdiction.
- Limit a resource-intensive task to run only on Engines whose processors are faster than a certain threshold, or that have more than a specified amount of memory or disk space.
- Direct a task that requires operating-system-specific resources to Engines that run that operating system.

Discriminator Conditions can be dynamically attached to a Service based on the Service Description on the Manager or set programmatically with the Driver.

Setting Discriminators in the Administration Tool

You can also attach Discriminators to Services in the GridServer Administration Tool, at **Services > Services > Service Conditions**. This page enables you to create Discriminators by entering three defining factors: the Services that the Discriminator affects, the types of Engines that can run on those Services, and optionally, if an EXTRACondition must be met. (See EXTRAConditions for more information about EXTRAConditions.) This differs from programmatic Discriminators because they aren't explicitly attached to a Service at its creation; instead, a group of Services is defined as being attached to that Discriminator, by Service name, application name, department name, or wildcards on that or other criteria.

By default, changes made to Discriminators in the GridServer Administration Tool are applied to Services immediately. This can be changed so that Discriminator changes only apply to subsequently created Services. To change this behavior, go to **Admin > System Admin > Manager Configuration > Services**, and under the **Scheduling**heading, change the value of **Apply Condition Admin Changes Immediately**.

Setting Discriminators Programmatically

You can use the GridServer API to set Discriminators for a Service or task. Create Discriminators with the `SchedulingConditionFactory`. In Java, this is located in `com.datasynapse.gridserver.client`. You can use the factory to create several Conditions. The method to use to create Discriminators is `createPropertyDiscriminator`.

The following example in Java code creates a `SchedulingConditionFactory`, which you then use to create a Discriminator Condition to run a Service or task only on Engines where the operating system is Linux:

```
SchedulingConditionFactory schedFactory =
SchedulingConditionFactory.getInstance();

Condition isLinux =
schedFactory.createPropertyDiscriminator(EngineProperties.OS,
SchedulingConditionFactory.CONTAINS, "Linux", false);
```

The `createPropertyDiscriminator` method takes four parameters:

- The first parameter is the name of an Engine property.
- The second parameter is a comparator used to compare the Engine property to the property value. You can use several comparators to compare numbers or strings. (See details in the `SchedulingConditionFactory` API documentation.) For example, the `CONTAINS`

comparator is true if an Engine property matches any one of a comma-delimited list of properties. `MATCHES` compares against a Java regular expression; `EQUAL` checks if parsed numerical values are equal.

- The third parameter is a property value.
- The fourth parameter defines what happens if the property is not set on an Engine. In this case, the Engine is not considered. In some situations, you might want the opposite behavior, such as when you are using a Discriminator to exclude a subset of Engines with a specific property, but allowing any other Engines.

After you create a Condition such as a Discriminator, you can use it when creating or submitting Services. The following example in Java code creates a Discriminator and sets it in a Service:

```
SchedulingConditionFactory schedFactory =
SchedulingConditionFactory.getInstance();

//Only run on Linux Engines

Condition isLinux =
schedFactory.createPropertyDiscriminator(EngineProperties.OS,
SchedulingConditionFactory.CONTAINS, "Linux", false);

Service s = ServiceFactory.getInstance().createService("MyService", null,
null, null, isLinux);
```

If you change the properties of Engines, the changes take effect during the execution of a Service. For example, if you configure a Discriminator attached to a running Service to look for a certain Engine property, changing this property can change what Engines work on that Service.

Discriminators are, however, attached to a Service at Service creation, so changes you make to the Discriminator affect only subsequently submitted Services, not Services that are already running.

PDriver Discriminators

When writing a PDS script, you can create job-level or task-level Discriminators to limit which Engines work on a PDriver job or task. The `discriminator` block specifies either a job-level or task-level Discriminator for a job.

For more information about PDriver Discriminators, see The Discriminator Block.

Affinity Conditions

When a Broker is assigning tasks to Engines, one of the methods used to make optimal matches is *affinity*, or the degree to which an Engine has initialization data and updates from a particular Service. If there are a number of possible matches at a point in the scheduling decision, the Broker assigns tasks to the match with the highest affinity score. Affinity is a number that is calculated between an Engine and a Service Session. By default, the affinity number is based on the amount of state the Engine has for the session, whether it has loaded its Grid libraries, and whether it is on a Home Broker.

It is possible to use *Affinity Condition* to add affinity, typically based on Engine properties. This Condition is similar to a Discriminator Condition, except when it is satisfied, it adds a defined number to the affinity score.

If you plan to use affinity, aside from Affinity Conditions, you also need to tune how and how much the scheduler uses affinity. For more information see the *TIBCO GridServer® Administration*.

Setting Affinity Conditions Programmatically

Create Property Affinity Conditions with the `SchedulingConditionFactory`, using the `createPropertyAffinity` method.

The following example in Java code creates a `SchedulingConditionFactory`, which is then used to create a Property Affinity Condition that would add four to the affinity of any Engine-Service pairing where the Engine has two CPUs:

```
SchedulingConditionFactory schedFactory =

     SchedulingConditionFactory.getInstance();

Condition dualAffinity =

     schedFactory.createPropertyAffinity(EngineProperties.CPU_NO,

     SchedulingConditionFactory.EQUALS, "2", 4.0);
```

Setting Affinity Conditions in the Administration Tool

You can also attach Property Affinity Conditions to Services in the GridServer Administration Tool. Go to **Services > Services > Service Conditions**. This page enables you to create Affinity Conditions by entering two defining factors: the Services that the Condition affects, and the affinity change that occurs based on an Engine's properties.

Task Affinity

Task Affinity provides the ability to run a set of Tasks on the same Engine or set of Engines. It is primarily used for data awareness and locality. For example, you might have a large dataset, and have Tasks that work on subsets of that dataset, so you would prefer that an Engine works on Tasks from the same subset.

Create Task Affinity Conditions with the `SchedulingConditionFactory`, using the `createTaskAffinity` method.

When set on a Task, it specifies that this Task must run only on an Engine that has already worked on a Task with the same `TaskAffinity` as specified by a tag.

The `wait` value indicates how long it must wait for an Engine with affinity before being allowed on any Engine. This wait countdown starts the first time that the task becomes available for scheduling, and only if it is the next task to be assigned to this set.

The first `TaskAffinity` for a given tag must be assigned a wait value of 0 so that it is immediately assigned to an Engine. Typically the remaining in the set would use the same value, for example, 10000 for a wait of 10s. An Engine retains the affinity for the duration of the Service Session but not beyond.

The tag is not guaranteed to be unique among all Services, so you would typically append a unique value to the tag, such as the Service ID. This is left to the caller, since there might be a unique string shorter than the Service ID which results in quicker string comparisons.

`TaskAffinity` can be used in a `ConditionSet`. For example, you can use this along with a `PropertyDiscriminator` and a `Descriptor` using an `AND` `ConditionSet`.

Custom Discriminator and Affinity Conditions

In addition to Conditions based on Engine Properties, you can implement your own custom Conditions. For example, you might have a number of Services that require a connection to a database and might have more Engines working on those Services than available connections. You could write a Server Hook that keeps track of how many tasks are currently running, and write a Custom Discriminator to prevent all Engines from taking a Task if all connections are in use by other Tasks.

To write a Custom Condition:

1. Implement the `CustomDiscriminator` or `CustomAffinity` interface in `SchedulingConditionFactory`.

2. Package your classes into a JAR file, and for each Broker on which this is used, place that file in `DS_MANAGER/webapps/livecluster/WEB-INF/lib`.
3. At Service creation time, create it using the `SchedulingConditionFactory.createCustom[Affinity/Discriminator]` call. Note that it is not necessary to have the JAR in a Java Driver classpath; likewise, this is also available for C++ and .NET Services, since the class is only used on Brokers.

Dependency Conditions

Dependency Conditions allow you to submit workflows to a Broker without an active Grid client to manage Dependencies (wait for completed tasks, submit more based on a successful outcome, and so on). When you submit a Session, you can require one or more tasks or entire Services to complete before the scheduled Service. These Dependencies can be Sessions or tasks already submitted, or ones that have not yet been created. This way, multiple tasks in different Services can be submitted, but they are not eligible for scheduling until certain conditions are met—namely the successful completion of specific tasks or Sessions.

Creating Dependencies

Dependencies are Conditions that are applied to a Service or a Task. This Condition has methods for adding Dependencies in the form of a Session ID, and for an optional Task ID. Dependencies also allow a Boolean operation that dictates whether to cancel an entire Task/Service when a dependent Task/Service fails.

You can create a forward Dependency for a Session, and optional Task ID, that does not yet exist. To create a forward Dependency, generate a reference ID to a Session, and then use that ID when creating the Session.

In Java, you use `com.datasynapse.gridserver.client.DependencyFactory` to create Dependency Conditions. You can assign more than one Dependency by using a `ConditionSet`. C++ and .NET APIs are similar. See the GridServer API documentation for more details.

If there is no session with the dependent Service ID on the Broker when the Session or task with Dependencies is added, it is automatically canceled because the Dependency does not exist. If it is a forward Dependency, no such cancellation is made.

You can specify that a Session or task be canceled if it has a Dependency that fails.

If a non-forward Dependency is made, and the session does not exist, the task is always canceled.

Administering Task Dependencies

You can view Dependencies in the GridServer Administration Tool. Select **Services > Services > Service Sessions**, or from the Task Admin page, available from the **Actions**list on the Service Session Admin page. On either page, select **Service Session Details**or **Task Details**from the **Actions**list. View the details for a list of Dependencies showing which are pending and which are complete.

You can remove Dependencies from a task or Service with the **Remove Dependencies**action on the **Actions**list on the Task Admin page. This removes the entire Dependency object, which removes all pending Dependencies; there is no a way to remove a single Dependency from the Administration Tool.

> **Note:**
>
> Task Dependencies are Broker-scope, and rely on Service and task events on a Broker. They do not work across Brokers.

> Because the default for the `PURGE_INVOCATION_DATA` option is `SERVICE_COMPLETED`, task information can be lost on a Service, making a dependent Service unavailable to find the information. In this situation, you can set `PURGE_INVOCATION_DATA` to `SERVICE_REMOVED` instead.

Queue Jump Conditions

The *Queue Jump Condition* is used to specify that a Task is placed at the front of the waiting queue, rather than at the back. It cannot be set on a Session. A typical use case is a Service that submits some work, waits for results, and submits more work based on earlier results. If an early task fails and you need to resubmit it, you use this condition to make sure it is executed as soon as possible instead of waiting for all other work to be complete.

Queue Jump Conditions are created programmatically using the `SchedulingConditionFactory` class. For more information about using this class, see Setting Discriminators Programmatically

Descriptor Conditions

Use Descriptor Conditions to display descriptive information about an invocation request.

Descriptive information about an invocation request is also available on the Task Admin page and Admin API. Also, the Descriptor name is part of any log message for a task with a Descriptor Condition.

Create Descriptors with the `DescriptorFactory`. For example:

```
DescriptorFactory descFactory = DescriptorFactory.getInstance();

descFactory.create("C++ Driver Linux Build");
```

You cannot create a Condition Set of Descriptors. A Descriptor Condition can be only a sole Condition in a Service or Service Set.

EXTRAConditions

An *EXTRACondition* is an External Resource Advisory Condition, and is used to prevent Engines from taking Tasks when a necessary external resource, such as a database connection, is not available. When using EXTRAConditions, the scheduler ensures that a task's required resources are available before assigning it to an Engine.

There are two parts involved in using EXTRAConditions: defining the Service Condition, and maintaining the EXTRACondition count. Like other Conditions, this type of Condition is defined on the Service Conditions page, and consists of one or more named resources with a numerical comparison. The current resource counts are then updated with a simple REST API.

Using the EXTRACondition REST Interface

Each EXTRACondition consists of a key and value pair that specifies a resource count. To use an EXTRACondition, you must create a key and then periodically update its count as the resource count changes.

Resource counts for EXTRAConditions are created and updated with a REST API.

The EXTRACondition REST Interface root is:

```
http://host:port/livecluster/restadmin/extracondition-admin/
```

The REST interface resides on the Primary Director. Values are backed up to the Secondary Director but updates to the Secondary Director are not replicated back if the Primary Director

comes back up. In the event of a Primary Director failure, you must use the Secondary Director's URL to update the EXTRACondition values until the Primary Director comes back up.

The API includes the following resources:

- `extracondition` — for operations on a single EXTRACondition
- `extraconditions` — for operations on all EXTRAConditions.

REST requests use basic authentication.

Creating and Updating an EXTRACondition

HTTP `PUT` requests are used to create and modify an EXTRAConditon. The following syntax is used:

```
/restadmin/extracondition-admin/extracondition/key
```

The value for the key must be passed in the body of the request. The request returns the value of the EXTRACondition as a string with an HTTP Response code of 201 upon success. Any other server-side error returns exception text with an HTTP Response code of 500.

For example, the following `PUT` request, when made with a value in the body of the request, creates a new EXTRACondition for the key `db.connections`, or updates the existing `db.connections` EXTRACondition:

```
/restadmin/extracondition-admin/extracondition/db.connections
```

Getting an Existing EXTRACondition

An HTTP `GET` request can be used to retrieve the value of an EXTRACondition. The following syntax is used:

```
/restadmin/extracondition-admin/extracondition/key
```

If the EXTRACondition is not found, an HTTP Response code of 404 is returned. Any other server-side error returns exception text with an HTTP Response code of 500.

For example, the following `GET` request returns the value of the EXTRACondition of the key `db.connections` as a string with an HTTP Response code of 200 upon success:

```
/restadmin/extracondition-admin/extracondition/db.connections
```

Deleting an existing EXTRACondition

An HTTP `DELETE` request can be used to remove an EXTRACondition. The following syntax is used:

```
/restadmin/extracondition-admin/extracondition/key
```

The request returns with an HTTP Response code of 200 even if the key does not exist. It only returns an exception text with an HTTP Response code of 500 for other server errors or an existing EXTRACondition was not able to be deleted.

For example, the following `DELETE` request deletes the EXTRACondition with the key `db.connections`.

```
/restadmin/extracondition-admin/extracondition/db.connections
```

Batch Operations

There is also an `extraconditions` resource that enables you to make an HTTP `GET` to retrieve all EXTRAConditions. The following syntax is used

```
/restadmin/extracondition-admin/extraconditions
```

The request returns a JSON array of key-value pairs formatted like the following:

```
{

    "Boston.BigFatDB.Connections":600,

    "SF.BigFatDB.Connections":500,

    "NYC.BigFatDB.Connections":1000

}
```

Similarly, an HTTP `PUT` with the same JSON in the body of the request updates or creates all pairs in the list.

To batch delete EXTRAConditions, you must use an HTTP `PUT` using the following syntax:

```
/restadmin/extracondition-admin/extraconditions/delete
```

The keys being deleted must be passed in the body of the request, like the following:

```
{

    "Boston.BigFatDB.Connections",

    "SF.BigFatDB.Connections",

    "NYC.BigFatDB.Connections"

}
```

Batch operation requests return with an HTTP Response code of 200 upon success. Any other server-side error returns exception text with an HTTP Response code of 500.

Setting EXTRAConditions

EXTRAConditions can be specified in Discriminators, which are attached to Services in the GridServer Administration Tool at **Services > Services > Service Conditions**. You can specify that one or more comparisons of EXTRACondition to resource criteria must be true, or the Task is not given to an Engine.

When creating a Discriminator using an EXTRACondition, there is a **modifier**column for each Condition, which you can set to a positive or negative number. When a Task is assigned to an Engine, the scheduler modifies the EXTRACondition's external resource value by adding the modifier value. Note that the external resource value is not automatically changed when a Task completes; you must script this using the REST interface, as described above.

Condition Sets

In addition to using a single Condition, you can also combine Conditions. A *Condition Set* is a set of one or more Conditions; the set is treated as a single Condition.

Create Condition Sets with `ConditionSetFactory`. (In Java, this is `com.datasynapse.gridserver.client.ConditionSetFactory`.) After you create a Condition Set, use its `add` method to add Conditions to it. There are three types of Condition Sets,

described below, which dictate how the Conditions are evaluated, and what Conditions can be added.

AND set

The AND set creates a set of ANDed Conditions. Two possible combinations of Conditions are legal in an AND set:

- An AND set can contain Discriminators, Dependencies, or both. If all the Conditions in the set are true, the Condition Set is true.
- Or, an AND set can contain Property Affinity Conditions. If all Affinity Conditions are true, the scores that match are added to the affinity score.

You cannot mix Discriminators, Dependencies, and Property Affinity Conditions in one AND set. Note that you cannot add Descriptor Conditions to an AND set.

OR Set

The OR set creates a set of Conditions for which any Condition can be true for the set to be true. The OR set can contain only Discriminators, Dependencies, and sets of either or both.

Service Set

The Service Set creates a Condition Set for adding different Condition types to an Invocation or Session. You can add this set only to `ServiceFactory.createService(...)`, `Service.execute(...)`, and `Service.execute(...)`. You can add only one Condition of each type (Discriminator/Dependency, Affinity, Descriptor, and QueueJump).

You can, however, add a Condition Set containing only one type of Condition. For example, a Service Set could include a Condition Set of Discriminators and Dependencies, a Condition Set of Property Affinity Conditions, and a Descriptor. (You cannot create a set of Descriptor Conditions.)

Engine Properties

Within GridServer, each Engine has a set of properties. GridServer sets some Engine properties automatically, such as the Engine’s operating system and the estimated speed of the Engine’s processor. You can also create custom properties for Engines.

Intrinsic Engine Properties

An Engine has a number of properties that reflect machine-specific information, such as OS, hostname, free memory, and so on. For information about these properties, see

`EngineProperties` in the `com.datasynapse.gridserver.engine` package of the API documentation.

Custom Engine Properties

You can create your own custom Engine properties, and give them values using the GridServer Administration Tool. To do so, first, create the property on the Manager, and then give it a value for each Engine, either from the GridServer Administration Tool or programmatically with the Admin API.

To create new custom Engine properties and give them values with the Administration Tool:

1. In the GridServer Administration Tool, go to **Grid Components > Engines > Engine Properties**.
2. In the upper right corner, click **Add**.
3. Enter a property name and a brief description, then click **Add**. You can now set a value to this property on any Engine.
4. Go to **Grid Components > Engines > Daemon Admin**and select **Set Property for Daemons on Page**or **Set Property for All Daemons**from the **Global Actions**list. This displays the user-defined properties you can set on Engines started by a Daemon.
5. Select one or more Engine Daemons from the list, then select a predefined property and enter a value, or enter a new property name and assign a value.

Instead of 3-4, you can also use the Admin API on a Driver to programmatically set them.

Engine Session Properties

Session Properties are properties that last for the duration of an Engine session on the Manager. They are set on an Engine when it logs in and reset when the Engine logs off.

This example sets a Session Property:

```
public void init() {

    // initialize some static data for use by another service

    EngineSession.setProperty("inited", "true");

    // this property can now be used by the Discriminator

    // of the other Service
```

```
}
```

See the API documentation for the `EngineSession` class for more information.

GPU Services Engine Properties

One set of intrinsic Engine properties is the CUDA GPU Engine properties set. On Windows and Linux systems, the Engine Daemon detects the presence and characteristics of GPU processors with the CUDA runtime library and provides those characteristics as Engine properties. This allows for the development of Services that can take advantage of machines that have GPU cards.

If the Engine Daemon detects a CUDA GPU, it sets the following Engine properties:

- `CUDA_DEVICES` – The number of CUDA devices detected on an Engine.
- `CUDA_FIRST_GPU_NAME` – The name of the first CUDA device.
- `CUDA_GLOBAL_MEMORY` – The amount of CUDA global memory supported. If there is more than one device, this is the minimum amount supported.
- `CUDA_VERSION` – The CUDA capability version. If there is more than one device, this is the minimum version.
- `CUDA_PROCESSORS` – The number of CUDA processors supported. If there is more than one device, this is the minimum number of processors.

Note that on 64-bit Windows machines, CUDA properties are only detected when the Engine Daemon is started on the physical console. Engine Daemons started via RDP do not detect any GPUs.

On Linux machines, you must ensure that the device files `/dev/nvidia*` exist and have read/write file permissions for the user running the Engine. This can be done by creating a startup script to load the driver kernel module and create the entries at boot time. For more information, see the Linux documentation at the NVIDIA support site.

See the CUDA example in the GridServer SDK for more information.

CUDA detection was tested on the GeForce 210 device with 3.0 drivers, GeForce 9800 GT device with 3.10 drivers, the Quadro NVS 295 with 3.0 and 3.20 drivers, and the Quadro FX 380 with 3.10 drivers.

MIC Processor Engine Properties

TIBCO GridServer® supports Intel Many Integrated Cores (MIC) coprocessors. This includes MIC detection and Engine properties that provide information that can be used for discrimination.

Intel MIC coprocessors, such as the Xeon PHI, are a multi-core processor architecture capable of running a large number of tasks in parallel due to a large number of physical cores available in the card. MIC coprocessors support two execution models: *native*, where programs are executed directly in the co-processor; and *offload*, where programs are executed in the host machine and some parts are executed in the coprocessor. Support for the MIC coprocessor in GridServer is restricted to the offload model.

Requirements

Support for MIC coprocessors requires the following:

- Linux on X86_64

  The following compilers are supported:

- Intel Composer XE 13

  MIC support has the following limitations:

- The Engine requires a pre-load of shared libraries to run Services with offload code.
- Services with offload code share global objects across Service instances. This is a limitation due to how Intel handles shared libraries with offload code.
- The availability of the MIC coprocessor is determined at Engine Daemon startup time. Changes to the status of coprocessors are not visible to the Engines.

Properties

If the Engine Daemon detects a MIC device, it sets the following Engine properties:

- `MICDevices` – The number of MIC devices enabled at the Engine Daemon startup time.
- `MICModel` – The identifier of the coprocessor model. This information is based on information from the `cpuid` opcode.
- `MICCores` – The number of logical cores.
- `MICMemory` – The amount of memory in bytes.

`MICModel`, `MICCores,` and `MICMemory` refer to the oldest co-processor available in the host machine. Oldest is defined based on this information from the `cpuid` opcode.

Configuration

To use MIC in your Service, you must do the following:

Engine Configuration

To preload the required shared libraries:

1. In the TIBCO GridServer® Administration Tool, go to **Grid Components > Engines> Engine Configurations**and select the appropriate Engine configuration.
2. Set this value under Engine Daemon and Instance Process Settings: Environment Variables:

   ```
   LD_PRELOAD=libjsig.so liboffload.so.5
   ```

   `libjsig.so` is part of the JRE and `liboffload.so.5` is part of the Intel runtime libraries.

Service Type

Services with offload code require `unloadNativeLibrary` set to false in the Service Type registry.

Grid Libraries

Services with the above Service Type setting share global objects across Services. Use the `<conflict>` setting in your `grid-library.xml` to avoid sharing of global objects with other Services:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library>

    ...

    <conflict>

        <grid-library-name>[name of grid-library that shares global
objects]</grid-library-name>

    </conflict>

    ...
```

```
</grid-library>
```

See the MIC example in the GridServer SDK for more information.

NUMA Engine Properties and Configuration

Non-uniform memory access (NUMA) can increase processor speed without increasing the load on the processor bus. In a NUMA system, CPUs are arranged in smaller systems called nodes. A node has its own processors and memory and is connected to the larger system. The system attempts to improve performance by scheduling processes on processors that are in the same node as the memory being used.

Requirements

NUMA is only supported on Windows Engines.

Windows 7 and Windows 2008 R2 require the following hotfix: http://support.microsoft.com/kb/2417038

32-bit Engine installations in 64-bit Windows are only able to address processors #0 through #31 in any given processor group.

Properties

The following Engine properties are related to NUMA support:

- `HIGHEST_NUMA_NODE` – The highest NUMA node available in the system. 0 is reported in non-NUMA systems.
- `PROCESS_SCHEDULING_POLICY` – The process scheduling policy in effect. The possible values are described below.

Configuration

To change the process scheduling policy to use NUMA:

1. In the TIBCO GridServer® Administration Tool, go to **Grid Components > Engines> Engine Configurations**and select the appropriate Engine configuration.
2. Change the Process Scheduling Policy to the desired value, which is described below.

There are three possible process scheduling policy settings in the Engine Configuration:

- **Native**(the default) uses the process scheduling as assigned by Windows.

- **Balanced**makes the best effort to balance the number of Engine processes and processors. The Engine process is allowed to run in all processors in a given processor group. Engines can run in processors from different NUMA nodes.
- **NUMA**makes the best effort to assign Engine processes to the ideal NUMA node. This option is implicitly 'balanced'. An Engine process is only allowed to run in processors in the ideal NUMA node.

Balanced Versus Native Policies

Windows assigns processes to processor groups at creation time in a round-robin fashion. This might be undesirable when you have an unbalanced number of logical processors in processor groups. Processor groups with fewer logical processors might end up with the same number of Engines as processor groups with more logical processors.

The balanced policy attempts to assign Engines proportionally to the number of logical processors in the processor groups. The processor group with more logical processors is assigned more Engines.

The native policy is preferred when the number of logical processors in processor groups is balanced.

Balanced and native policies behave identically in machines with at most one processor group or no support for processor groups.

Extending GridServer

You can extend the GridServer Manager and Engine with Manager and Engine Hooks. A Manager Hook enables you to interface your own Java object directly with the Manager’s event processing mechanism and interact with any Server Event. An Engine Hook can perform user-defined operations on Engine startup or termination, or before or after Service invocation.

A Hook consists of two parts: the class implementation of the Hook, and the Hook registration. Register Manager Hooks on the Hook Admin page. Register Engine Hooks with an XML file that is deployed to Engines.

Manager Hooks

You can create Manager Hooks on the Broker or the Director. Depending on where the hook resides, it receives a different subset of the Server Events broadcast by the Manager. See the `ServerEvents` class JavaDoc for available events and the components for which they are

relevant. For example, a hook that needs to see the addition or removal of an Engine must run on the Broker, because the `ENGINE_ADDED` and `ENGINE_REMOVED` events are Broker specific.

For details on Manager Hook implementation, see the JavaDoc documentation for the `ServerHook` class. After implementing a Manager Hook, contain its class definition in a JAR file in the shared classes directory (`hooks/`*component*, where *component* is either `broker` or `director`).

To register a Manager Hook in the GridServer Administration Tool, go to **Admin > System Admin > Manager Hooks**. The Hook Admin page enables you to edit, enable, or disable hooks on the Manager. To add a new hook, select **Create New Hook**from the Global Actions list. This opens a Hook Editor in a new window. Enter a filename for the hook XML file, and select whether to apply the hook to the Director or Broker. Enter the name of a class in the hooks directory and click Update Properties to display an updateable property list. After you enter properties, click **Save**to edit the hook or **Cancel**to revert to the last saved version of the hook.

From the Actions control of each hook, you can Enable, Disable, Edit, or Delete an existing hook. Note that you do not need to restart the Manager after deploying the JAR file. However, if you redeploy a JAR file, you must remove and re-add the hook for any new changes to take effect.

Engine Hooks

Engine Hooks are implemented to perform operations on an Engine:

- On library initialization
- Before or after Service invocation
- On Engine termination

Engine Hooks are useful for performing AOP operations such as setting up a global environment, administering a database or other external component, or collecting metrics on performance or utilization without requiring Services to call a library on their own. You can also use them to do things like restart an Engine after a certain number of invocations.

Engine Hook implementation details are in the JavaDoc documentation for the `EngineHook` class.

Add Engine Hooks by adding the XML and JAR file containing the class definition to a Grid Library. In order for an Engine Hook to always be loaded at Engine startup, you can define it as a Super Grid Library. For more information about Super Grid Libraries, see the *TIBCO GridServer® Administration*. You can use multiple XML files in Grid Libraries (as opposed to the method of

having a single `hooks.xml` file, used in previous releases). Note that if you are using Grid Libraries, hooks in the `deploy` directory do not work. For an example of the XML format to use for your Hook, see the `EngineHook` JavaDoc.

Engine Hook Example

The following example initializes a JDBC database.

```
package examples.hook;

import com.datasynapse.gridserver.engine.*;

import java.sql.*;

import java.util.*;

/*

 * This is an example of a hook that initializes data from a database.

 * The property "initialized" can be used to discriminate on Services, so
that

 * only Engines that have initialized the data will take tasks.

 */

public class JDBCHook extends EngineHook {

    public void initialized() {

        initializeData();

        EngineSession.setProperty("initialized", "true");

    }

    // static method is used by the task

    public static Vector getData() {

        return vData;
```

```
    }
 private void initializeData() throws ClassNotFoundException {
        System.out.println("initializing");
        Class cl = Class.forName(getDriver());
        System.out.println("Driver class:" + cl);
 boolean successful = false;
        do {
        try {
                Connection conn = DriverManager.getConnection(getUrl(),
                        getUsername(), getPassword());
                PreparedStatement ps = conn.prepareStatement("select *
from
                    people");
                ResultSet rs = ps.executeQuery();
                System.out.println("rs:" + rs);
                while ( rs.next() )
                    vData.add( rowToLine(rs) );
                successful = true;
            } catch (SQLException e) {
                System.out.println("JDBCHook: failed to retrieve data,
will try
                                    again.");
            }
```

```
            if (!successful) {

                try { Thread.sleep(getFrequency()); } catch
(InterruptedException ie) { break; }

            }

} while (!successful);

    }

static String rowToLine( ResultSet input ) throws SQLException {

        StringBuffer buf = new StringBuffer();

        int cols = input.getMetaData().getColumnCount();

        for ( int i=1; i <= cols; i++ ) {

            buf.append(input.getString(i));

            buf.append(' ');

        }

 buf.append('\n');

        return buf.toString();

    }

    public final void setUrl(String url) {

        _url = url;

    }

    public final String getUrl() {

        return _url;

    }
```

```
    public final String getDriver() {
        return _driver;
    }
public final void setDriver(String driver) {
        _driver = driver;
    }
 public final String getUsername() {
        return _user;
    }
    public final void setUsername(String user) {
        _user = user;
    }
    public final String getPassword() {
        return _pass;
    }
 public final void setPassword(String password) {
        _pass = password;
    }
    public final void setFrequency(long frequency) {
        _frequency = frequency;
    }
    public final long getFrequency() {
```

```
        return _frequency;

    }

    private String _url;

    private String _driver;

    private String _user;

    private String _pass;

    private long _frequency = 5000;

    private static Vector vData = new Vector();

}
```

The following is also an example of the XML to add to the `hooks.xml` file for the JDBC example given above.

```
<hook class="examples.hook.JDBCHook">

        <property name="username" value="sa"/>

        <property name="password" value=""/>

        <property name="url"

            value="jdbc:HypersonicSQL:hsql://%server%:2034"/>

        <property name="driver" value="org.hsql.jdbcDriver"/>

</hook>
```

Implementing Engine Hooks as a Grid Library

To implement an Engine Hook as a Grid Library you must create a type of Grid Library called a Super Grid Library. If you are unfamiliar with implementing a Grid Library, review Creating Services.

Super Grid Libraries are very similar to regular Grid Libraries. You still need to use a ZIP archive to contain the code and XML documents just like a regular Grid Library. In the `grid-library.xml`, you must add an attribute to the `<grid-library>` element called `super` which is set to `true`. Additionally, for the code to be used as an Engine Hook, you must specify a folder where the XML descriptor for the hook is contained. Below is an example of a `grid-library.xml` file that has been set up for an Engine Hook:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library super="true">

    <grid-library-name>AddPropHook</grid-library-name>

    <grid-library-version>1.0</grid-library-version>

    <jar-path>

        <pathelement>jars</pathelement>

    </jar-path>

    <hooks-path>

        <pathelement>hook</pathelement>

    </hooks-path>

</grid-library>
```

Populating the Super Grid Library is also very similar to a regular Grid Library. The folder structure includes the folder you specified in the `grid-library.xml` for the hook's XML descriptor and the `grid-library.xml` is placed at the top level of the folder structure within the Grid Library. For example, the following is the top level folder structure for an example Engine Hook:

```
    jars/

    AddPropHook.xml

    grid-library.xml
```

Once the Super Grid Library has been constructed, upload it to the Manager in exactly the same way as a standard Grid Library. After it is deployed, the Engines discover it is a Super Grid Library and immediately execute the `initialized()` method.

Task Instrumentation

This Appendix describes the instrumentation phases produced by enabling task instrumentation.

Overview

This Appendix describes the instrumentation phases produced by enabling task instrumentation. To enable task instrumentation, see the *TIBCO GridServer® Administration*.

> **Note:**
>
> Use task instrumentation only for development, not for production environments. Task instrumentation slows down the Manager significantly, and also requires additional disk space, so it is important to disable it after you finish using it.

Syntax

All instrumentation phases have an absolute time marker, which is the time at the start of the action. Actions also have a relative duration marker, if it is possible to measure the duration. The times are marked according to the client's clock.

Instrumentation phases have the following syntax:

```
[Client] [Action] [Object]
```

Client

The Client of an instrumentation phase can be one of the following:

- Engine
- Driver
- Broker

Action

The Action of an instrumentation phase can be one of the following:

Instrumentation Phase Actions

ACTION	DESCRIPTION
Send	A send of a message. The absolute time is the start time of the send, and there is no duration value.
Receive	A receive of a message. The absolute time is the end of the retrieval. There is no duration value.
Retrieve	A receive, with a measurement of the duration. The absolute time is the time at which the retrieval started.
Serialize	The conversion of an in-memory object to its serialized format, for transfer to another client.
Deserialize	The conversion of a serialized object to an in-memory object.
Write	The writing of data to a file, typically for DDT (Direct Data Transfer).
Download	The downloading of data from another client.
Call	A call to a user-implemented method.
Load	A native library load.

Object

The Object of an instrumentation phase can be one of the following

Instrumentation Phase Objects

OBJECT	DESCRIPTION
Jar	The JAR file, which is only used for dynamic class loading.
Instance	The instance object, which is either the task or the initialization data.

OBJECT	DESCRIPTION
Input	The input data or message.
Output	The output data or message.
Update	The update data, message, or call.
Checkpoint	Checkpoint data, if checkpointing is enabled.
Library	A native library.
Initialize	The initialization call.
Service	The Service call.
Completed	The callback on completion.
Failed	The callback on failure.
Serialize	The call to a user-implemented native serialize method.
Deserialize	The call to a user-implemented native deserialize method.

Phases

The following is the complete list of all phases.

Driver-side

The following are Driver-side phases:

Instrumentation Driver-side Phases

PHASE	DESCRIPTION
Driver Serialize Jar	The serialization of the JAR file when the JAR file is set.
Driver Serialize Instance	The serialization of the Service instance object.
Driver Serialize Input	The serialization of the Service input.

PHASE	DESCRIPTION
Driver Send Input	The time the Driver sends the input message to the Broker. Keep in mind that more than one input can be sent in one message.
Driver Call Completed	The callback of a successful task.
Driver Call Failed	The callback of a failed task.
Driver Download Output:	The download of output over DDT.
Driver Deserialize Output	
Driver Retrieve Output	The time at which the Driver receives the output message from the Broker. Keep in mind that more than one output can be retrieved in one message.

Engine-side

The following are Engine-side phases:

Instrumentation Engine-side Phases

PHASE	DESCRIPTION
Engine Receive Input	The time at which the Engine receives the input message from the Broker.
Engine Deserialize Instance	The deserialization of the Service instance object.
Engine Call Initialize	The initialization call.
Engine Download Update	The download of update data.
Engine Deserialize Update	The deserialization of update data.
Engine Call Update	The update call.
Engine Deserialize Input	The conversion of the serialized input to an in-memory object.

PHASE	DESCRIPTION
Engine Download Checkpoint	The download of checkpoint data from another Engine.
Engine Call Service	The Service call.
Engine Serialize Output	The serialization of the output.
Engine Send Output	The time at which the Engine sends the output message to the Broker.

Broker-side

The following are Broker-side phases:

Instrumentation Broker-side Phases

PHASE	DESCRIPTION
Broker Receive Input	The time at which the Broker received the input from the Driver.
Broker Send Input	The time at which the Broker sent the input to the Engine.
Broker Receive Output	The time at which the Broker received the output from the Engine.
Broker Send Output	The time at which the Broker sent the output to the Driver.
Broker Remove Output	The time at which the Broker removed the output due to the acknowledgment from the Driver.

DDT file write

The following are DDT file write phases:

DDT File Write Phases

PHASE	DESCRIPTION
Write Input:	The input file write.
Write Output:	The output file write.
Write Instance	The instance object write.
Write Jar	The JAR file write.
Write Update	The update data write.

Client can refer to the Engine, Driver, or Broker.

Native

The following are native phases:

Native Phases

PHASE	DESCRIPTION
Engine Load Library	The load of a native dynamic library.
Driver Call Serialize	The native object serialize call.
Driver Call Deserialize	The native object deserialize call.

Example Phases in a Service Execution

The following is a reference example of a typical list of native Service execution phases. Not all possible phases are shown.

Example Service Execution Phases

PHASE	DESCRIPTION
Driver Serialize Instance	The serialization of the Service instance object.
Driver Serialize Input	The serialization of the Service input.
Driver Write Input	Driver writes input data to the driver machine file system.

PHASE	DESCRIPTION
Driver Write Instance	Driver writes task data to the driver machine file system.
Driver Send Input	Driver sends the input message to the Broker. Keep in mind that more than one input can be sent in one message.
Broker Receive Input:	Broker receives the new task entry.
Broker Send Input	Broker sends the input to the Engine.
Engine Receive Input	Engine receives the input message from the Broker.
Engine Deserialize Instance	Engine deserializes task data.
Engine Load Library	Engine loads all necessary libraries for the task.
Engine Call Initialize	Engine calls init method.
Engine Deserialize Input	Engine deserializes input data.
Engine Call Service	Engine executes the task.
Engine Serialize Output	Engine serializes output data.
Engine Write Output	Engine writes output data to disk.
Engine Send Output	Broker receives the task complete message from Engine.
Broker Receive Output	Broker marks the task complete.
Broker Send Output	Broker notifies the driver that the task is ready to collect.
Driver Retrieve Output	Driver retrieves completed tasks info from Broker.
Driver Download Output	Driver gets output data from Engine.

PHASE	DESCRIPTION
Driver Deserialize Output	Driver deserializes output data.
Driver Call Completed	Driver completes the task.
Broker Remove Output	Broker removes the task from the task entry.

The grid-library.dtd

The `grid-library.xml` configuration file in the root of a Grid Library must be a well-formed XML file. The GridServer SDKs include a `grid-library.dtd` file that can be used to validate the XML file. The DTD is also shown in this section

The grid-library.dtd

The following is the `grid-library.dtd` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- Copyright 2022 TIBCO Software, Inc. All Rights Reserved. -->

<!-- Grid-Library is in the root of the GL. -->

<!ELEMENT grid-library (grid-library-name, grid-library-version?,
arguments?, dependency*, conflict*, jar-path*, lib-path*,

assembly-path*, command-path*, hooks-path*, environment-variables*, java-
system-properties*)>

<!ATTLIST grid-library jre (true|false) "false">

<!ATTLIST grid-library bridge (true|false) "false">

<!ATTLIST grid-library super (true|false) "false">

<!ATTLIST grid-library os CDATA #IMPLIED >

<!ATTLIST grid-library compiler CDATA #IMPLIED >

<!-- The library name. -->
```

```
<!ELEMENT grid-library-name (#PCDATA)>

<!-- The version. If not specified, 0 is implied. -->

<!ELEMENT grid-library-version (#PCDATA)>

<!-- Additional arguments to the JVM. -->

<!ELEMENT arguments (property*)>

<!-- A library dependency. Dependencies can be specified by package name
and

     optional version.

     If the version is not specified, the latest version is chosen at load
time.  -->

<!ELEMENT dependency (grid-library-name, grid-library-version?)>

<!-- A library conflict. Indicates that this library conflicts with the
given

     library.

     If this library is NOT a dependency, and grid-library-name="*",

     then it indicates that this library conflicts with all other
libraries

     (aside from its own dependencies).   -->

<!ELEMENT conflict (grid-library-name)>

<!-- The JAR path. If specified, all jars and classes in the path are
loaded. -->

<!ELEMENT jar-path (pathelement*)>

<!ATTLIST jar-path os CDATA #IMPLIED>

<!ATTLIST jar-path compiler CDATA #IMPLIED>
```

```
<!-- An element of a path, typically a directory.  -->

<!ELEMENT pathelement (#PCDATA)>

<!-- Load library path. If not specified, it is assumed that no native
libraries are

     loaded by this GL.

     If this is specified and it the library was not loaded at init time,
the Engine      will restart, adding this path to the current path. -->

<!ELEMENT lib-path (pathelement*)>

<!ATTLIST lib-path os CDATA #IMPLIED>

<!ATTLIST lib-path compiler CDATA #IMPLIED>

<!-- .NET assembly path.
System.AppDomain.CurrentDomain.AppendPrivatePath(path) will be called on
this path,

    which add it to the lookup location for assemblies.  -->

<!ELEMENT assembly-path (pathelement*)>

<!ATTLIST assembly-path os CDATA #IMPLIED>

<!ATTLIST assembly-path compiler CDATA #IMPLIED>

<!-- The path in which the Engine will search for Command Service
executables.  -->

<!ELEMENT command-path (pathelement*)>

<!ATTLIST command-path os CDATA #IMPLIED>

<!ATTLIST command-path compiler CDATA #IMPLIED>

<!-- Engine hooks library path. Hook will be initialized as libraries are
loaded. -->

<!ELEMENT hooks-path (pathelement*)>
```

```
<!ATTLIST hooks-path os CDATA #IMPLIED>

<!ATTLIST hooks-path compiler CDATA #IMPLIED>

<!-- Environment variables to set. Environment variables are set via JNI
immediately prior to executing a task using this library. -->

<!ELEMENT environment-variables (property*)>

<!ATTLIST environment-variables os CDATA #IMPLIED>

<!ATTLIST environment-variables compiler CDATA #IMPLIED>

<!-- A property, used by env vars & system props. -->

<!ELEMENT property (name,value)>

<!-- The name for a property element. -->

<!ELEMENT name (#PCDATA)>

<!-- The value for a property element. -->

<!ELEMENT value (#PCDATA)>

<!-- Java system properties, which are set upon load. -->

<!ELEMENT java-system-properties (property*)>

<!ATTLIST java-system-properties os CDATA #IMPLIED>

<!ATTLIST java-system-properties compiler CDATA #IMPLIED>

<!-- end of grid-library dtd -->
```

REST API Reference

Web services that can be used by Grid Server components such as Director, Broker, Engine, and Driver are explained here. By using these web services, you can perform different types of operations over the grid components.

APIs for the following classes are available:

- BatchAdmin
- BrokerAdmin
- DriverAdmin
- DriverManager
- EngineAdmin
- EngineDaemonAdmin
- ManagerAdmin
- ServiceAdmin
- UserAdmin
- Version

BatchAdmin

Provides administrative access to the Batches and Batch executions on a Broker. Methods in class BatchAdmin are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
batch-definition	POST	Stores the XML contents to a Batch definition on the Broker.
batch-definition	DELETE	Removes a Batch definition stored on the Broker.
all-batch-execution-info	GET	Retrieves all the Batch execution information stored on the Broker.
all-batch-info	GET	Retrieves information about all the Batch entries on the Broker.
batch-count	GET	Retrieves the total number of Batch entries stored on the Broker.

METHOD	METHOD TYPE	DESCRIPTION
batch-definition	GET	Retrieves XML contents of the Batch definition.
batch-definition-names	GET	Retrieves the Batch definition names stored on the Broker.
batch-execution-count	GET	Retrieves the total number of Batch executions stored on the Broker.
batch-execution-ids	GET	Retrieves the list of Batch execution IDs stored on the Broker.
batch-execution-info	GET	Retrieves information about a given Batch execution ID stored on the Broker.
batch-execution-info-by-batch-id	GET	Retrieves all the Batch execution information about a given Batch entry stored on the Manager.
batch-ids	GET	Retrieves the list of Batch entry IDs stored on the Broker.
batch-info	GET	Retrieves the Batch entry information about a given Batch entry ID stored on the Broker.
running-batch-execution-count	GET	Retrieves the total number of running batch executions stored on the Broker.
scheduled-batch-count	GET	Retrieves the total number of scheduled Batch entries on the Broker.

METHOD	METHOD TYPE	DESCRIPTION
selected-batch-execution-info	POST	Retrieves information about given Batch execution IDs stored on the Broker.
selected-batch-info	POST	Retrieves information about given Batch entry IDs stored on the Broker.
available	GET	Retrieves whether methods are available.
batch	DELETE	Removes the information about a finished or suspended Batch entry.
batch-execution	DELETE	Removes the information about a finished or failed Batch execution.
finished-batch-executions	DELETE	Removes the information about all finished Batch executions on the Broker.
finished-batches	DELETE	Removes the information about all finished Batch entries on the Broker.
resume-batch	PUT	Resumes a suspended Batch entry.
schedule-batch-definitions	GET	Runs the specified Batch definition.
suspend-all-batches	PUT	Suspends all Batch entries running on the Broker.
suspend-batch	PUT	Suspends a scheduled Batch entry.

batch-definition

Stores the XML contents to a Batch definition on the Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/batch/batch-definition
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
name	String	Name of Batch definition	NewCalculatorServiceExample
contents	String	XML contents of Batch definition	<?xml version="1.0" encoding <Batch class="com.datasynaps gridserver.batch.Batch"> <property name="name" value= <property name="description" <property name="type" value= <Schedule class="com.datasyn gridserver.batch.Schedule"> <property name="type" value= </Schedule> <Command class="com.datasyna gridserver.batch.command. LogCommand"><property name=" </Command> <ServiceRunnerReference clas

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			gridserver.batch. ServiceRunnerReference"><pro </ServiceRunnerReference> <Command class="com.datasyna gridserver.batch.command. LogCommand"><property name=" </Command> </Batch>

Example Response

`204 no content`

Result: The Batch definition is added to the Broker.

batch-definition

Removes a Batch definition stored on the Broker.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/batch/batch-definition
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of batch definition whose definition needs to be removed from the Broker	`CalculatorServiceExample`

Example Response

`204 no content`

Result: Batch definition is removed from the Broker.

all-batch-execution-info

Retrieves all the Batch execution information stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/all-batch-execution-info
```

Example Response

```
[
    {
        "name": "CalculatorServiceExample",
        "startTime": 1578568486582,
        "status": "Failed",
        "batchId": 4562207508471809452,
        "batchExecutionId": 6976777355170490963
    }
```

```
]
```

all-batch-info

Retrieves information about all the Batch entries on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/all-batch-info
```

Example Response

```
[
    {
        "name": "CalculatorServiceExample",
        "type": "serial",
        "definition": "CalculatorServiceExample.xml",
        "description": "Batch Example that runs the Calculator Service",
        "suspended": false,
        "status": "Finished",
        "batchId": 4562207508471809452,
        "nextRuntime": null,
        "scheduleType": "Immediate",
        "localFileName": "CalculatorServiceExample.xml",
        "submitTime": 1578568485925
    }
```

```
]
```

batch-count

Retrieves the total number of Batch entries stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-count
```

Example Response

0

batch-definition

Retrieves XML contents of the Batch definition.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-definition
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of batch definition whose XML content needs to be retrieved	`CalculatorServiceExample`

Example Response

```
<?xml version="1.0" encoding="UTF-8"?>

<Batch class="com.datasynapse.gridserver.batch.Batch">

<property name="name" value="CalculatorServiceExample" />
```

```
<property name="description" value="Batch Example that runs the Calculator
Service" />

<property name="type" value="serial" />

<Schedule class="com.datasynapse.gridserver.batch.Schedule"><property
name="type" value="Immediate" />

</Schedule>

<Command
class="com.datasynapse.gridserver.batch.command.LogCommand"><property
name="message" value="Starting Java Calculator Service Example" />

</Command>

<ServiceRunnerReference
class="com.datasynapse.gridserver.batch.ServiceRunnerReference"><property
name="name" value="JavaCalculatorRunnerExample" /></
ServiceRunnerReference>

<Command
class="com.datasynapse.gridserver.batch.command.LogCommand"><property
name="message" value="Done running Java Calculator Service Example" />

</Command>

</Batch>
```

batch-definition-names

Retrieves the Batch definition names stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-definition-names
```

Example Response

```
[

    "CalculatorServiceExample.xml"

]
```

batch-execution-count

Retrieves the total number of Batch executions stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-execution-count
```

Example Response

0

batch-execution-ids

Retrieves the list of Batch execution IDs stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-execution-ids
```

Example Response

```
[

    6976777355170490963

]
```

batch-execution-info

Retrieves information about a given Batch execution ID stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-execution-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
instanceId	Long	Batch execution ID of the Batch whose information needs to be retrieved	4534714832987845913

Example Response

```
{

    "name": "CalculatorServiceExample",

    "startTime": 1578902422107,

    "batchExecutionId": 4534714832987845913,

    "status": "Finished",

    "batchId": 7270548312726024832

}
```

batch-execution-info-by-batch-id

Retrieves all the Batch execution information about a given Batch entry stored on the Manager.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-execution-info-
by-batch-id
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
batchId	Long	Batch entry ID of the Batch whose execution information needs to the retrieved	4562207508471809452

Example Response

```
[
    {
        "name": "CalculatorServiceExample",
        "startTime": 1578568486582,
        "status": "Failed",
        "batchId": 4562207508471809452,
        "batchExecutionId": 6976777355170490963
    }
]
```

batch-ids

Retrieves the list of Batch entry IDs stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-ids
```

Example Response

```
[

    4562207508471809452

]
```

batch-info

Retrieves the Batch entry information about a given Batch entry ID stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/batch-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
batchId	Long	Batch entry ID of the Batch whose batch entry information needs to the retrieved	4562207508471809452

Example Response

```
{

    "name": "CalculatorServiceExample",

    "type": "serial",

    "definition": "CalculatorServiceExample.xml",

    "description": "Batch Example that runs the Calculator Service",
```

```
    "suspended": false,
    "status": "Finished",
    "batchId": 4562207508471809452,
    "nextRuntime": null,
    "scheduleType": "Immediate",
    "localFileName": "CalculatorServiceExample.xml",
    "submitTime": 1578568485925
}
```

running-batch-execution-count

Retrieves the total number of running batch executions stored on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/running-batch-execution-count
```

Example Response

0

scheduled-batch-count

Retrieves the total number of scheduled Batch entries on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/scheduled-batch-count
```

Example Response

0

selected-batch-execution-info

Retrieves information about given Batch execution IDs stored on the Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/batch/selected-batch-execution-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of Batch Execution IDs whose information needs to be retrieved	`[ 3933318490 9810832, 4534714832 987845913]`

Example Response

```
[

    {

        "name": "CalculatorServiceExample",

        "startTime": 1578900113522,

        "batchExecutionId": 39333184909810832,

        "status": "Finished",

        "batchId": 5838911030065021273

    },

    {
```

```
        "name": "CalculatorServiceExample",
        "startTime": 1578902422107,
        "batchExecutionId": 4534714832987845913,
        "status": "Finished",
        "batchId": 7270548312726024832
    }
]
```

selected-batch-info

Retrieves information about given Batch entry IDs stored on the Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/batch/selected-batch-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of Batch Entry IDs whose information needs to be retrieved	`[ 5838911030065021273, 7270548312726024832]`

Example Response


```
[
    {
        "name": "CalculatorServiceExample",
        "type": "serial",
        "definition": "CalculatorServiceExample.xml",
        "suspended": false,
        "description": "Batch Example that runs the Calculator Service",
        "status": "Finished",
        "submitTime": 1578900113129,
        "batchId": 5838911030065021273,
        "nextRuntime": null,
        "scheduleType": "Immediate",
        "localFileName": "CalculatorServiceExample.xml"
    },
    {
        "name": "CalculatorServiceExample",
        "type": "serial",
        "definition": "CalculatorServiceExample.xml",
        "suspended": false,
        "description": "Batch Example that runs the Calculator Service",
        "status": "Finished",
```

```
        "submitTime": 1578902422059,
        "batchId": 7270548312726024832,
        "nextRuntime": null,
        "scheduleType": "Immediate",
        "localFileName": "CalculatorServiceExample.xml"
    }
]
```

available

Retrieves whether methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/available
```

Example Response

`True` or `False`

batch

Removes the information about a finished or suspended Batch entry.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/batch/batch
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
batchIds	Long	Batch Entry ID of Batch whose information needs to be removed	926167698331434208

Example Response

`204 no content`

Result: Information about the finished or suspended Batch entry is removed.

batch-execution

Removes the information about a finished or failed Batch execution.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/batch/batch-execution
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
instanceId	Long	Execution ID of Batch whose information needs to be removed	39333184909810832

Example Response

`204 no content`

Result: Information about the finished or failed Batch execution is removed.

finished-batch-executions

Removes the information about all finished Batch executions on the Broker.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/batch/finished-batch-executions
```

Example Response

`204 no content`

Result: Information about all finished Batch executions is removed from the Broker.

finished-batches

Removes the information about all finished Batch entries on the Broker.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/batch/finished-batches
```

Example Response

`204 no content`

Result: Information about all finished Batch entries on the Broker is removed.

resume-batch

Resumes a suspended Batch entry.

Example Request

```
PUT http://example.com:8080/livecluster/rest/batch/resume-batch
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`bacthId`	Long	Batch Entry ID of the Batch that needs to be resumed	`5838911030065021273`

Example Response

`204 no content`

Result: The suspended Batch entry is resumed.

schedule-batch-definitions

Runs the specified Batch definition.

Example Request

```
GET http://example.com:8080/livecluster/rest/batch/schedule-batch-definition
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the batch definition that must be run	`CalculatorServiceExample`

Example Response

`3668132062277102496`.

Result: The running Batch Definition ID is returned.

suspend-all-batches

Suspends all Batch entries running on the Broker.

Example Request

```
PUT http://example.com:8080/livecluster/rest/batch/suspend-all-batches
```

Example Response

`204 no content`

Result: All Batch entries running on the Broker are suspended.

suspend-batch

Suspends a scheduled Batch entry.

Example Request

```
PUT http://example.com:8080/livecluster/rest/batch/suspend-batch
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`batchId`	Long	Batch entry ID of the Batch that needs to be suspended	`5838911030065021273`

Example Response

`204 no content`

Result: The scheduled Batch entry is suspended.

BrokerAdmin

BrokerAdmin APIs provide administrative access to Brokers on a Director. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
service-discriminator	POST	Adds the service discriminator.
service-discriminator	DELETE	Deletes the Service Discriminator.
all-broker-info	GET	Retrieves information about all Brokers logged in to the Director.
broker-count	GET	Retrieves the total number of Brokers logged in to the Director.
broker-info	GET	Retrieves information about a given Broker.
engine-router	GET	Retrieves the Engine Router assigned to this Broker.
service-discriminator	GET	Retrieves the PropertyConditionSet used by the Service Discriminator.
service-discriminator-names	GET	Retrieves the names of all Service Discriminators.
shared-brokers	GET	Retrieves Broker sharing.
available	GET	Retrieves whether the methods are available.
driver-weight	PUT	Sets the Driver distribution weighing relative to other Brokers.

METHOD	METHOD TYPE	DESCRIPTION
engine-router	POST	Sets an Engine router for a given Broker.
engine-weight	PUT	Sets the Engine distribution weighting relative to other Brokers.
maximum-engines	PUT	Sets the maximum number of Engines the Broker can manage.
min-idle-home-engines	PUT	Sets the minimum number of Idle Home Machines that the Broker is left to manage.
minimum-engines	PUT	Sets the minimum number of Engines the Broker is left to manage.
shared-brokers	PUT	Sets the Broker sharing.

service-discriminator

Adds the service discriminator.

Example Request

```
POST http://example.com:8080/livecluster/rest/broker/service-discriminator
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the Service Discriminator that needs to be added	`TestDiscriminatorNew`

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Data of Service Discriminator that needs to be added	{
			"descriptionConditio
			"sets": [],
			"conditions": [
			{
			"name": "ser
			"comparison"
			"value": "Li
			"nullCompare
			}
			],
			"type": 0
			},
			"engineCondition":{
			"sets": [],
			"conditions": [
			{
			"name": "os"
			"comparison"
			"value": "li
			"nullCompare

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			}], "type": 0 } }

Example Response

`204 no content`

Result:A new Service Discriminator is added.

service-discriminator

Deletes the Service Discriminator.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/broker/service-discriminator
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the Service Discriminator that needs to be deleted	`TestDiscriminatorNew`

Example Response

`204 no content`

Result: The specified Service Discriminator is deleted.

all-broker-info

Retrieves information about all Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/all-broker-info
```

Example Response


```
[
        {
                "minIdleHomeEngines": 0,
                "name": "abcd",
                "baseUrl": "http://win64vm091:8000/livecluster",
                "engineWeight": 1.0,
                "driverWeight": 1.0,
                "maxEngines": 2500,
                "minEngines": 0,
                "brokerId": 899860077,
                "driverCount": 1,
                "failover": false,
                "engineCount": 0,
                "busyEngineCount": 0,
```

```
                "engineRoutingConditions": [
                        {
                                "name": "AMIAvailZone",
                                "comparison": 1,
                                "value": "fvszgvsa",
                                "nullCompare": false
                        }
                ],
                "engineRoutingComparators": [
                        {
                                "methodNum": 0,
                                "value": "fvszgvsa",
                                "name": "AMIAvailZone",
                                "nullComparison": false
                        }
                ],
                "driverRoutingComparators": null,
                "driverRoutingConditions": null,
                "hostname": "http://win64vm091:8000/livecluster"
        }
]
```

broker-count

Retrieves the total number of Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/broker-count
```

Example Response

`1`.

broker-info

Retrieves information about a given Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/broker-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker whose information needs to be retrieved	`899860077`

Example Response

```
{

    "minIdleHomeEngines": 0,

        "name": "abcd",

        "baseUrl": "http://win64vm091:8000/livecluster",
```

```
        "engineWeight": 1.0,
        "driverWeight": 1.0,
        "maxEngines": 2500,
        "minEngines": 0,
        "brokerId": 899860077,
        "driverCount": 1,
        "failover": false,
        "engineCount": 0,
        "busyEngineCount": 0,
        "engineRoutingConditions": [
                {
                        "name": "AMIAvailZone",
                        "comparison": 1,
                        "value": "fvszgvsa",
                        "nullCompare": false
                }
        ],
        "engineRoutingComparators": [
                {
                        "methodNum": 0,
                        "value": "fvszgvsa",
                        "name": "AMIAvailZone",
```

```
                        "nullComparison": false
                }
        ],
        "driverRoutingComparators": null,
        "driverRoutingConditions": null,
        "hostname": "http://win64vm091:8000/livecluster"
}
```

engine-router

Retrieves the Engine Router assigned to this Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/engine-router
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker for which information of assigned Engine Router needs to be retrieved	`899860077`

Example Response


```
{

    "sets": null,

    "conditions": [

        {

            "name": "AMIAvailZone",

            "comparison": 1,

            "value": "fvszgvsa",

            "nullCompare": false

        }

    ],

    "type": 0

}
```


service-discriminator

Retrieves the PropertyConditionSet used by the Service Discriminator.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/service-discriminator
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of Service Discriminator for which PropertyConditionSet needs to be retrieved	`TestDiscriminator`

Example Response


```
[
        {
                "sets": [],
                "conditions": [
                        {
                                "name": "serviceName",
                                "comparison": 1,
                                "value": "Lin",
                                "nullCompare": false
                        }
                ],
                "type": 0
        },
        {
                "sets": [],
```

```
                "conditions": [
                        {
                                "name": "os",
                                "comparison": 3,
                                "value": "linux64",
                                "nullCompare": false
                        }
                ],
                "type": 0
        },
        {
                "sets": [],
                "conditions": [],
                "type": 0
        }
]
```

service-discriminator-names

Retrieves the names of all Service Discriminators.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/service-discriminator-names
```

Example Response

```
[

   "testdiscriminator"

]
```

shared-brokers

Retrieves Broker sharing.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/shared-brokers
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
id	Long	Broker ID of Broker for which Broker sharing needs to be retrieved	899860077

Example Response

```
1
```

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/broker/available
```

Example Response

`True` or `False`

driver-weight

Sets the Driver distribution weighing relative to other Brokers.

Example Request

```
PUT http://example.com:8080/livecluster/rest/broker/driver-weight
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker for which Driver distribution weighting needs to be set	`1961263499`
`weight`	Double	Driver weight	`2`

Example Response

`204 no content`

Result: The driver-weight value is updated for the corresponding Broker.

engine-router

Sets an Engine router for a given Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/broker/engine-router
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker for which Engine Router needs to be set	`1580108171`

—	JSON	Engine Router data	{
			"sets": null,
			"conditions": [
			{
			"name": "os",
			"comparison": 1,
			"value": "win64"
			"nullCompare": f
			},
			{
			"name": "osName"
			"comparison": 1,
			"value": "WIN2K1
			"nullCompare": f
			}
			],

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			"type": 0 }

Example Response

`204 no content`

Result: Engine router is set for the specified Broker.

engine-weight

Sets the Engine distribution weighting relative to other Brokers.

Example Request

```
PUT http://example.com:8080/livecluster/rest/broker/engine-weight
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker for which Engine Distribution weighting needs to be set	`1961263499`
`weight`	Double	Engine weight	`2`

Example Response

`207 no content`

Result: Engine weight is updated for the specified Broker.

maximum-engines

Sets the maximum number of Engines the Broker can manage.

Example Request

```
PUT http://example.com:8080/livecluster/rest/broker/maximum-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker	`1580108171`
`maxEngines`	Int	Maximum number of Engines that the given Broker can manage	`6`

Example Response

`207 no content`

Result: The maximum number of Engines that the Broker can Manage is set.

min-idle-home-engines

Sets the minimum number of Idle Home Machines that the Broker is left to manage.

Example Request

```
PUT http://example.com:8080/livecluster/rest/broker/min-idle-home-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker	`1580108171`
`minEngines`	Int	Minimum number of Idle Home Engines on the given Broker	`1`

Example Response

`204 no content`

Result: The minimum number of Idle Home Engines for the Broker is set.

minimum-engines

Sets the minimum number of Engines the Broker is left to manage.

Example Request

```
PUT http://example.com:8080/livecluster/rest/broker/minimum-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker	`1580108171`
`minEngines`	Int	Minimum number of Engines that the given Broker can manage	`5`

Example Response

`204 no content`

Result: The minimum number of Engines for the Broker is set.

shared-brokers

Sets the Broker sharing.

Example Input

```
PUT http://example.com:8080/livecluster/rest/broker/shared-brokers
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Broker ID of Broker	`1580108171`
`brokerSharing`	String	Broker ID that needs to be shared	`1961263499`

Example Response

```
204 no content
```

Result: Broker sharing is set.

DriverAdmin

Provides administrative access to the Drivers on a Broker. Methods in class DriverAdmin are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
all-driver-info	GET	Retrieves information about all Drivers logged in to the Broker.
driver-count	GET	Retrieves the total number of Drivers logged in to the Broker.

METHOD	METHOD TYPE	DESCRIPTION
driver-info	GET	Retrieves information about Drivers with a given hostname.
available	GET	Retrieves whether the methods are available.

all-driver-info

Retrieves information about all Drivers logged in to the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/driver/all-driver-info
```

Example Response

```
{

        "properties": [

                {

                        "name": "Username",

                        "value": "internal"

                },

                {

                        "name": "maxPriority",

                        "value": "None"

                },

                {

                        "name": "Hostname",
```

```
                "value": "10.128.88.91"
        },
        {
                "name": "roles",
                "value": "internal"
        },
        {
                "name": "logToSystem",
                "value": "true"
        },
        {
                "name": "baseDir",
                "value": "F:/datasynapse/manager-data"
        },
        {
                "name": "ClientJavaVersion",
                "value": "7.1.1.xxxxxx"
        },
        {
                "name": "Version",
                "value": "7.1.1.xxxxxx"
        }
```

```
        ],
        "username": "internal",
        "serviceCount": 0,
        "loginDate": 1578562190177,
        "hostname": "10.128.88.91"
}
```

driver-count

Retrieves the total number of Drivers logged in to the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/driver/driver-count
```

Example Response

```
1
```

driver-info

Retrieves information about Drivers with a given hostname.

Example Request

```
GET http://example.com:8080/livecluster/rest/driver/driver-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`hostname`	String	Name of the host whose driver	`10.128.88.91`

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
		information needs to be retrieved	

Example Response


```
{
        "properties": [
            {
                "name": "Username",
                "value": "internal"
            },
            {
                "name": "maxPriority",
                "value": "None"
            },
            {
                "name": "Hostname",
                "value": "10.128.88.91"
            },
            {
                "name": "roles",
                "value": "internal"
            },
```

```
            {
                "name": "logToSystem",
                "value": "true"
            },
            {
                "name": "baseDir",
                "value": "F:/datasynapse/manager-data"
            },
            {
                "name": "ClientJavaVersion",
                "value": "7.1.1.xxxxxx"
            }
            {
                "name": "Version",
                "value": "7.1.1.xxxxxx"
            }
        ],
        "username": "internal",
        "serviceCount": 0,
        "loginDate": 1578562190177,
        "hostname": "10.128.88.91"
```

```
}
```

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/driver/available
```

Example Response

`True` or `False`

EngineAdmin

EngineAdmin APIs provide administrative access to Engines on a Broker. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
all-engine-info	GET	Retrieves information about all Engines logged in to the Broker.
busy-engine-count	GET	Retrieves the total number of Engines logged in to the Broker and are currently executing a task.
engine-count	GET	Retrieves the total number of Engines logged in to the Broker.
engine-ids	GET	Retrieves the list of Engine IDs logged in to the Broker.
engine-info	GET	Retrieves information about a given Engine.
engine-info-by-properties	POST	Retrieves information about Engines logged in to the

METHOD	METHOD TYPE	DESCRIPTION
		Broker that match the Engine Condition.
log-url-list	GET	Retrieves the log file URLs on a given Engine when the Engine logging is enabled on the Server.
selected-engine-info	POST	Retrieves information, which includes all instances for the given IDs, about given Engines logged in to the Broker.
available	GET	Retrieves whether the methods are available.
kill-all-engines	PUT	Logs out all the Engines from the Broker.
kill-engine	PUT	Logs out a given Engine from the Broker.
park-engines	POST	Sets Engine's parked flag to true for passed Engine IDs.
unpark-engines	POST	Sets Engine's parked flag to false for passed Engine IDs.
parked-engines	GET	Retrieves the list of Engine IDs that have the parked flag set to true.

all-engine-info

Retrieves information about all Engines logged in to the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/all-engine-info
```

Example Response


```
{

        "instance": "0",

        "properties": [

                {

                        "name": "AMIProductCode",

                        "value": "null"

                },

                {

                        "name": "dataDir",

                        "value": "C:\\TIBCO\\DataSynapse\\Engine\\data\
\win64vm091-0\\ddt"

                },

                {

                        "name": "availableDiskInMB",

                        "value": "12503261184"

                },

                {

                        "name": "cpuIdString",

                        "value": "GenuineIntel"

                },

                {
```

```
                        "name": "osVersion",
                        "value": "6.1"
                },
                {
                        "name": "workUrl",
                        "value": "http://10.128.88.91:27159/work/
win64vm091-0"
                },
                {
                        "name": "QuarantineStatus",
                        "value": "New Engine"
                },
                {
                        "name": "CUDAGlobalMemory",
                        "value": "null"
                },
                {
                        "name": "osUsername",
                        "value": "SYSTEM"
                },
                {
                        "name": "CUDAProcessors",
```

```
                        "value": "null"

                },

                {

                        "name": "CUDAFirstGPUName",

                        "value": "null"

                },

                {

                        "name": "dataUrl",

                        "value": "http://10.128.88.91:27159/data//
win64vm091-0/ddt/"

                },

                {

                        "name": "additionalPlatforms",

                        "value": "null"

                },

                {

                        "name": "MICMemory",

                        "value": "null"

                },

                {

                        "name": "ClientJavaVersion",

                        "value": "7.1.1.xxxxxx"
```

```
        },
        {
                "name": "homeBrokers",
                "value": ""
        },
        {
                "name": "configurationName",
                "value": "win64:default"
        },
        {
                "name": "AMIInstanceId",
                "value": "null"
        },
        {
                "name": "cpuSocketCount",
                "value": "1"
        },
        {
                "name": "IP",
                "value": "10.128.88.91"
        },
        {
```

```
                "name": "totalMemInKB",
                "value": "2096696"
            },
            {
                "name": "lastUpdated",
                "value": "Mon Jan 06 06:54:35 PST 2020"
            },
            {
                "name": "guid",
                "value": "deadbeef8091"
            },
            {
                "name": "RTLDGlobal",
                "value": "null"
            },
            {
                "name": "CUDADevices",
                "value": "0"
            },
            {
                "name": "GridLibs",
                "value": "python-win64-
```

```
vc14,3.7.0;SpeedLink,7.0.0.165519;pybridge-win64-vc14,7.1"

                },

                {

                        "name": "multicore",

                        "value": "false"

                },

                {

                        "name": "os",

                        "value": "win64"

                },

                {

                        "name": "MICModel",

                        "value": "null"

                },

                {

                        "name": "NETFrameworkVersions",

                        "value": "4.5,4.5FullProfile"

                },

                {

                        "name": "MICCores",

                        "value": "null"

                },
```

```
        {
                "name": "username",
                "value": "win64vm091"
        },
        {
                "name": "GridLibUpdateTime",
                "value": "1578322476609"
        },
        {
                "name": "id",
                "value": "4848394891525229394"
        },
        {
                "name": "freeMemInKB",
                "value": "163244"
        },
        {
                "name": "cpuNo",
                "value": "1"
        },
        {
                "name": "pid",
```

```
                "value": "792"
            },
            {
                "name": "cpuCoreCount",
                "value": "1"
            },
            {
                "name": "AMIAvailZone",
                "value": "null"
            },
            {
                "name": "cpuMHz",
                "value": "2597.0"
            },
            {
                "name": "CUDAVersion",
                "value": "null"
            },
            {
                "name": "cpuTotal",
                "value": "2794.447000"
            },
```

```
        {
                "name": "Version",
                "value": "7.1.1.xxxxxx"
        },
        {
                "name": "freeDiskInMB",
                "value": "12503261184"
        },
        {
                "name": "MICDevices",
                "value": "null"
        },
        {
                "name": "instance",
                "value": "0"
        },
        {
                "name": "crtVersions",
                "value": "VC12,VC14,VC15,VC16"
        },
        {
                "name": "HighestNumaNode",
```

```
                "value": "0"

        },

        {

                "name": "homeDir",

                "value": "C:\\TIBCO\\DataSynapse\\Engine"

        },

        {

                "name": "ProcessSchedulingPolicy",

                "value": "native"

        },

        {

                "name": "NETFramework",

                "value": "true"

        },

        {

                "name": "gridLibrary",

                "value": "null"

        },

        {

                "name": "cpuThreadCount",

                "value": "1"

        },
```

```
                {
                        "name": "osName",
                        "value": "WIN2K12"
                }
        ],
        "username": "win64vm091",
        "engineId": 4848394891525229394,
        "serviceId": -1,
        "invocationId": -1,
        "busy": false,
        "elapsedTime": -1
}
```


busy-engine-count

Retrieves the total number of Engines logged in to the Broker and are currently executing a task.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/busy-engine-count
```

Example Response

`1`.

Result: The number of Engines that are busy is returned.

engine-count

Retrieves the total number of Engines logged in to the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/engine-count
```

Example Response

`1`.

Result: The total number of Engines that are logged in to the Broker is returned.

engine-ids

Retrieves the list of Engine IDs logged in to the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/engines/engine-ids
```

Example Response

```
[

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394,

    4848394891525229394
```

```
]
```

engine-info

Retrieves information about a given Engine.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/engine-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
id	Long	ID of Engine whose information needs to be retrieved	4848394891525229394
instance	String	Engine instance whose information needs to be retrieved	1

Example Response

```
{
    "instance": "0",
    "properties": [
        {
            "name": "AMIProductCode",
            "value": "null"
```

```
        },
        {
            "name": "dataDir",
            "value": "C:\\TIBCO\\DataSynapse\\Engine\\data\\win64vm091-0\
\ddt"
        },
        {
            "name": "availableDiskInMB",
            "value": "11030"
        },
        {
            "name": "cpuIdString",
            "value": "GenuineIntel"
        },
        {
            "name": "osVersion",
            "value": "6.1"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.91:27159/work/win64vm091-0"
        },
```

```
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "CUDAGlobalMemory",
            "value": "null"
        },
        {
            "name": "osUsername",
            "value": "SYSTEM"
        },
        {
            "name": "CUDAProcessors",
            "value": "null"
        },
        {
            "name": "CUDAFirstGPUName",
            "value": "null"
        },
        {
            "name": "dataUrl",
```

```
            "value": "http://10.128.88.91:27159/data//win64vm091-0/ddt/"
        },
        {
            "name": "additionalPlatforms",
            "value": "null"
        },
        {
            "name": "MICMemory",
            "value": "null"
        },
        {
            "name": "ClientJavaVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "homeBrokers",
            "value": ""
        },
        {
            "name": "configurationName",
            "value": "win64:default"
        },
```

```
        {
            "name": "AMIInstanceId",
            "value": "null"
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "IP",
            "value": "10.128.88.91"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "lastUpdated",
            "value": "Wed Jan 08 23:08:10 PST 2020"
        },
        {
            "name": "guid",
```

```
            "value": "deadbeef8091"

        },

        {

            "name": "RTLDGlobal",

            "value": "null"

        },

        {

            "name": "CUDADevices",

            "value": "0"

        },

        {

            "name": "GridLibs",

            "value": "python-win64-
vc14,3.7.0;SpeedLink,7.0.0.165519;pybridge-win64-vc14,7.1"

        },

        {

            "name": "multicore",

            "value": "false"

        },

        {

            "name": "os",

            "value": "win64"
```

```
        },
        {
            "name": "MICModel",
            "value": "null"
        },
        {
            "name": "NETFrameworkVersions",
            "value": "4.5,4.5FullProfile"
        },
        {
            "name": "MICCores",
            "value": "null"
        },
        {
            "name": "username",
            "value": "win64vm091"
        },
        {
            "name": "GridLibUpdateTime",
            "value": "1578553692407"
        },
        {
```

```
            "name": "id",
            "value": "4848394891525229394"
        },
        {
            "name": "freeMemInKB",
            "value": "1019692"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "pid",
            "value": "5012"
        },
        {
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "AMIAvailZone",
            "value": "null"
```

```
        },
        {
            "name": "cpuMHz",
            "value": "2597.0"
        },
        {
            "name": "CUDAVersion",
            "value": "null"
        },
        {
            "name": "cpuTotal",
            "value": "1676.7"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "freeDiskInMB",
            "value": "11030"
        },
        {
```

```
            "name": "MICDevices",
            "value": "null"
        },
        {
            "name": "instance",
            "value": "0"
        },
        {
            "name": "crtVersions",
            "value": "VC12,VC14,VC15,VC16"
        },
        {
            "name": "HighestNumaNode",
            "value": "0"
        },
        {
            "name": "homeDir",
            "value": "C:\\TIBCO\\DataSynapse\\Engine"
        },
        {
            "name": "ProcessSchedulingPolicy",
            "value": "native"
```

```
        },
        {
            "name": "NETFramework",
            "value": "true"
        },
        {
            "name": "gridLibrary",
            "value": "null"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "osName",
            "value": "WIN2K12"
        }
    ],
    "username": "win64vm091",
    "engineId": 4848394891525229394,
    "serviceId": -1,
    "invocationId": -1,
```

```
    "busy": false,
    "elapsedTime": -1
}
```

engine-info-by-properties

Retrieves information about Engines logged in to the Broker that match the Engine Condition.

Example Request

```
POST http://example.com:8080/livecluster/rest/engine/engine-info-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine condition data depending on which Engines are retrieved	{ "sets": null, "conditions": [{ "name": "IP", "comparison": 1, "value": "10.128 "nullCompare": f }, {

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			"name": "os", "comparison": 1, "value": "win64" "nullCompare": f }], "type": 0 }

Example Response

```
[
    {
        "instance": "0",
        "properties": [
            {
                "name": "AMIProductCode",
                "value": "null"
            },
            {
                "name": "dataDir",
                "value": "C:\\TIBCO\\DataSynapse\\Engine\\data\
```

```
\win64vm142-0\\ddt"

            },

            {

                "name": "availableDiskInMB",

                "value": "13793"

            },

            {

                "name": "cpuIdString",

                "value": "GenuineIntel"

            },

            {

                "name": "osVersion",

                "value": "6.1"

            },

            {

                "name": "workUrl",

                "value": "http://10.128.88.142:27159/work/win64vm142-0"

            },

            {

                "name": "QuarantineStatus",

                "value": "New Engine"

            },
```

```
            {
                "name": "CUDAGlobalMemory",
                "value": "null"
            },
            {
                "name": "osUsername",
                "value": "SYSTEM"
            },
            {
                "name": "CUDAProcessors",
                "value": "null"
            },
            {
                "name": "CUDAFirstGPUName",
                "value": "null"
            },
            {
                "name": "dataUrl",
                "value": "http://10.128.88.142:27159/data//win64vm142-0/
ddt/"
            },
            {
```

```
            "name": "additionalPlatforms",
            "value": "null"
        },
        {
            "name": "MICMemory",
            "value": "null"
        },
        {
            "name": "ClientJavaVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "homeBrokers",
            "value": ""
        },
        {
            "name": "configurationName",
            "value": "win64:default"
        },
        {
            "name": "AMIInstanceId",
            "value": "null"
```

```
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "IP",
            "value": "10.128.88.142"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "lastUpdated",
            "value": "Tue Jan 14 05:37:32 EST 2020"
        },
        {
            "name": "guid",
            "value": "deadbeef8142"
        },
        {
```

```
                "name": "RTLDGlobal",
                "value": "null"
            },
            {
                "name": "CUDADevices",
                "value": "0"
            },
            {
                "name": "GridLibs",
                "value": "calculator,1.0.0.1;python-win64-
vc14,3.7.0;SpeedLink,7.0.0.165519;pybridge-win64-vc14,7.1"
            },
            {
                "name": "multicore",
                "value": "false"
            },
            {
                "name": "os",
                "value": "win64"
            },
            {
                "name": "MICModel",
```

```
            "value": "null"
        },
        {
            "name": "NETFrameworkVersions",
            "value": "3.5,3.5SP1,4,4FullProfile"
        },
        {
            "name": "MICCores",
            "value": "null"
        },
        {
            "name": "username",
            "value": "win64vm142"
        },
        {
            "name": "GridLibUpdateTime",
            "value": "1578998255589"
        },
        {
            "name": "id",
            "value": "6153431282294490153"
        },
```

```
        {
            "name": "freeMemInKB",
            "value": "1356648"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "pid",
            "value": "2072"
        },
        {
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "AMIAvailZone",
            "value": "null"
        },
        {
            "name": "cpuMHz",
```

```
            "value": "2597.0"
        },
        {
            "name": "CUDAVersion",
            "value": "null"
        },
        {
            "name": "cpuTotal",
            "value": "2704.3"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "freeDiskInMB",
            "value": "13793"
        },
        {
            "name": "MICDevices",
            "value": "null"
        },
```

```
        {
            "name": "instance",
            "value": "0"
        },
        {
            "name": "crtVersions",
            "value": "VC12,VC14,VC15,VC16"
        },
        {
            "name": "HighestNumaNode",
            "value": "0"
        },
        {
            "name": "homeDir",
            "value": "C:\\TIBCO\\DataSynapse\\Engine"
        },
        {
            "name": "ProcessSchedulingPolicy",
            "value": "native"
        },
        {
            "name": "NETFramework",
```

```
            "value": "true"
        },
        {
            "name": "gridLibrary",
            "value": "null"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "osName",
            "value": "WIN2K12"
        }
    ],
    "username": "win64vm142",
    "engineId": 6153431282294490153,
    "serviceId": -1,
    "invocationId": -1,
    "busy": false,
    "elapsedTime": -1
```

```
    }
```

log-url-list

Retrieves the log file URLs on a given Engine when the Engine logging must is enabled on the Server.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/log-url-list
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine ID for which logs need to be retrieved	`4848394891525229394`
`instance`	String	Instance ID for corresponding Engine ID	`0`

Example Response

```
[

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.08.2020-23.08.11.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.54.36.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.54.18.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
```

```
engine-0.01.06.2020-06.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.53.42.log",

    "http://10.128.88.91:27159/work/win64vm091-1/log/
engine-1.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-2/log/
engine-2.01.08.2020-22.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-3/log/
engine-3.01.08.2020-22.53.52.log",

    "http://10.128.88.91:27159/work/win64vm091-4/log/
engine-4.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-5/log/
engine-5.01.08.2020-22.53.52.log",

    "http://10.128.88.91:27159/work/win64vm091-6/log/
engine-6.01.08.2020-22.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-7/log/
engine-7.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-8/log/
engine-8.01.08.2020-22.53.55.log",

    "http://10.128.88.91:27159/work/win64vm091-9/log/
engine-9.01.08.2020-22.53.52.log"

]
```

selected-engine-info

Retrieves information, which includes all instances for the given IDs, about given Engines logged in to the Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/engine/selected-engine-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine IDs of Engines that are logged in to the Broker	[4848394891525229394,4848394

Example response


```
{

        "instance": "0",

        "properties": [

            {

                "name": "AMIProductCode",

                "value": "null"

            },

            {

                "name": "dataDir",

                "value": "C:\\TIBCO\\DataSynapse\\Engine\\data\
\win64vm091-0\\ddt"

            },

            {

                "name": "availableDiskInMB",

                "value": "11566219264"
```

```
        },
        {
            "name": "cpuIdString",
            "value": "GenuineIntel"
        },
        {
            "name": "osVersion",
            "value": "6.1"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.91:27159/work/win64vm091-0"
        },
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "CUDAGlobalMemory",
            "value": "null"
        },
        {
```

```
                "name": "osUsername",
                "value": "SYSTEM"
            },
            {
                "name": "CUDAProcessors",
                "value": "null"
            },
            {
                "name": "CUDAFirstGPUName",
                "value": "null"
            },
            {
                "name": "dataUrl",
                "value": "http://10.128.88.91:27159/data//win64vm091-0/
ddt/"
            },
            {
                "name": "additionalPlatforms",
                "value": "null"
            },
            {
                "name": "MICMemory",
```

```
            "value": "null"
        },
        {
            "name": "ClientJavaVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "homeBrokers",
            "value": ""
        },
        {
            "name": "configurationName",
            "value": "win64:default"
        },
        {
            "name": "AMIInstanceId",
            "value": "null"
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
```

```
        {
            "name": "IP",
            "value": "10.128.88.91"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "lastUpdated",
            "value": "Wed Jan 08 23:08:10 PST 2020"
        },
        {
            "name": "guid",
            "value": "deadbeef8091"
        },
        {
            "name": "RTLDGlobal",
            "value": "null"
        },
        {
            "name": "CUDADevices",
```

```
                "value": "0"

            },

            {

                "name": "GridLibs",

                "value": "python-win64-
vc14,3.7.0;SpeedLink,7.0.0.165519;pybridge-win64-vc14,7.1"

            },

            {

                "name": "multicore",

                "value": "false"

            },

            {

                "name": "os",

                "value": "win64"

            },

            {

                "name": "MICModel",

                "value": "null"

            },

            {

                "name": "NETFrameworkVersions",

                "value": "4.5,4.5FullProfile"
```

```
        },
        {
            "name": "MICCores",
            "value": "null"
        },
        {
            "name": "username",
            "value": "win64vm091"
        },
        {
            "name": "GridLibUpdateTime",
            "value": "1578553692407"
        },
        {
            "name": "id",
            "value": "4848394891525229394"
        },
        {
            "name": "freeMemInKB",
            "value": "1019692"
        },
        {
```

```
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "pid",
            "value": "5012"
        },
        {
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "AMIAvailZone",
            "value": "null"
        },
        {
            "name": "cpuMHz",
            "value": "2597.0"
        },
        {
            "name": "CUDAVersion",
            "value": "null"
```

```
        },
        {
            "name": "cpuTotal",
            "value": "1676.7"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "freeDiskInMB",
            "value": "11566219264"
        },
        {
            "name": "MICDevices",
            "value": "null"
        },
        {
            "name": "instance",
            "value": "0"
        },
        {
```

```
            "name": "crtVersions",
            "value": "VC12,VC14,VC15,VC16"
        },
        {
            "name": "HighestNumaNode",
            "value": "0"
        },
        {
            "name": "homeDir",
            "value": "C:\\TIBCO\\DataSynapse\\Engine"
        },
        {
            "name": "ProcessSchedulingPolicy",
            "value": "native"
        },
        {
            "name": "NETFramework",
            "value": "true"
        },
        {
            "name": "gridLibrary",
            "value": "null"
```

```
            },
            {
                "name": "cpuThreadCount",
                "value": "1"
            },
            {
                "name": "osName",
                "value": "WIN2K12"
            }
    ],
        "username": "win64vm091",
        "engineId": 4848394891525229394,
        "serviceId": -1,
        "invocationId": -1,
        "busy": false,
        "elapsedTime": -1
}
```


available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/available
```

Example Response

`True` or `False`

kill-all-engines

Logs out all the Engines from the Broker.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engine/kill-all-engines
```

Example Response

`204 no content`

Result: All Engines are logged out of the Broker.

kill-engine

Logs out a given Engine from the Broker.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engine/kill-engine
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	ID of the Engine that needs to be logged out	`4848394891525229394`
`instance`	String	Engine instance that needs to be logged out	`0`

Example Response

`204 no content`

Result: The specified Engine is logged out of the Broker.

park-engines

Sets Engine's parked flag to true for passed Engine IDs.

Example Request

```
POST http://example.com:8080/livecluster/rest/engine/park-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of Engine Daemon IDs for which the parked flag needs to be set to true	[6153431282294490153, 3574664343712327623]

Example Response

`204 no content`

Result: Specified Engine Daemons are parked.

unpark-engines

Sets Engine's parked flag to false for passed Engine IDs.

Example Request

```
POST http://example.com:8080/livecluster/rest/engine/unpark-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of Engine Daemon IDs for which the parked flag needs to be set to false	[6153431282294490153]

Example Response

`204 no content`

Result: Specified Engine Daemons are unparked.

parked-engines

Retrieves the list of Engine IDs that have the parked flag set to true.

Example Request

```
GET http://example.com:8080/livecluster/rest/engine/parked-engines
```

Example Response

```
[
    "9007929106493580894"
]
```

EngineDaemonAdmin

EngineDaemonAdmin APIs provide administrative access to the Engine Daemons on a Director. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
all-engine-daemon-info	GET	Retrieves information about all the Engine Daemons logged in to the Director.
default-properties	GET	Retrieves the names and descriptions from the default Engine property list.
engine-daemon-count	GET	Retrieves the total number of Engine Daemons logged in to the Director.
engine-daemon-ids	GET	Retrieves the list of Engine Daemon IDs logged in to the Director.
engine-daemon-info	GET	Retrieves information about the given Engine Daemon logged in to the Director.
engine-daemon-info-by-properties	POST	Retrieves information about all the Engine Daemons logged in to the Director that match the Engine Condition.
log-url-list	GET	Retrieves the log file URLs from the given Engine Daemon when Engine Logging is enabled on the Server.
selected-engine-daemon-info	POST	Retrieves information about a given Engine Daemon logged in to the Director.
available	GET	Retrieves whether the methods are available.
default-property	DELETE	Removes the default property.

METHOD	METHOD TYPE	DESCRIPTION
property	DELETE	Removes a property from the Engine Daemon.
property-by-properties	DELETE	Removes a property from selected Engine Daemons that match the Engine Condition.
restart-engine-daemon	PUT	Forces the Engine Daemon to log off and restart.
restart-engine-daemon-by-properties	POST	Forces those Engine Daemons that match the Engine Condition to log off and restart.
all-enabled	PUT	Sets whether all Engine Daemons can run Engines.
all-start-mode	PUT	Sets whether all Engines have to be started manually or automatically.
configuration	PUT	Sets the configuration to run on the Engines.
configuration-by-properties	POST	Sets the configuration to run on the Engines that match the Engine Condition.
default-property	PUT	Creates a new default Engine Property and its description.
directors	PUT	Sets the primary and secondary Directors for the Daemons, overriding the values in the configuration.
directors-by-properties	POST	Sets the primary and secondary Directors on the Daemons for the Daemons that match the Engine

METHOD	METHOD TYPE	DESCRIPTION
		condition, overriding the values in the configuration.
enabled	PUT	Sets whether the Engine Daemon can run Engines.
enabled-by-properties	POST	Sets whether the Engine Daemon that matches the Engine Condition can run Engines.
instances	PUT	Sets the number of Engine instances to run, overriding the value in the configuration.
instances-by-properties	POST	Sets the number of Engine instances to run, overriding the value in the configuration for Engines that match the Engine Conditions.
property	PUT	Sets a property on the Engine Daemon.
property-by-properties	POST	Sets a property on selected Engine Daemons that match the Engine Condition.
start-mode	PUT	Sets whether the Engines have to be started manually or automatically.
start-mode-by-properties	POST	Sets whether the Engines that match the Engine Condition have to be started manually or automatically.

all-engine-daemon-info

Retrieves information about all the Engine Daemons logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/all-engine-daemon-info
```

Example Response


```
{

        "properties": [

                {

                        "name": "ClientPlatform",

                        "value": "win64"

                },

                {

                        "name": "cpuCoreCount",

                        "value": "1"

                },

                {

                        "name": "availableDiskInMB",

                        "value": "10965"

                },

                {

                        "name": "IP",

                        "value": "10.128.88.91"

                },
```

```
        {
                "name": "numInstances",
                "value": "10"
        },
        {
                "name": "QuarantineStatus",
                "value": "New Engine"
        },
        {
                "name": "username",
                "value": "win64vm091"
        },
        {
                "name": "workUrl",
                "value": "http://10.128.88.91:27159/work"
        },
        {
                "name": "freeMemInKB",
                "value": "125520"
        },
        {
                "name": "ClientEngineDaemonVersion",
```

```
                "value": "7.1.1.xxxxxx"
        },
        {
                "name": "Version",
                "value": "7.1.1.xxxxxx"
        },
        {
                "name": "os",
                "value": "win64"
        },
        {
                "name": "timeStamp",
                "value": "1578558278135"
        },
        {
                "name": "cpuUtilization",
                "value": "98"
        },
        {
                "name": "cpuSocketCount",
                "value": "1"
        },
```

```
        {
                "name": "id",
                "value": "4848394891525229394"
        },
        {
                "name": "freeDiskInMB",
                "value": "10965"
        },
        {
                "name": "dsCpuUtilization",
                "value": "2"
        },
        {
                "name": "cpuNo",
                "value": "1"
        },
        {
                "name": "guid",
                "value": "deadbeef8091"
        },
        {
                "name": "totalMemInKB",
```

```
                "value": "2096696"
            },
            {
                "name": "cpuThreadCount",
                "value": "1"
            },
            {
                "name": "Timestamp",
                "value": "1578553686304"
            },
            {
                "name": "osName",
                "value": "WIN2K12"
            },
            {
                "name": "cpuTotal",
                "value": "1676.7"
            },
            {
                "name": "instance",
                "value": "0"
            },
```

```
                {
                        "name": "osUsername",
                        "value": "SYSTEM"
                }
        ],
        "enabled": true,
        "engineId": 4848394891525229394,
        "instances": 10,
        "lastLoginDate": "Jan 8, 2020 11:08:06 PM",
        "installDate": "2020-01-06 06:53:25.803",
        "autoStart": false,
        "manualStart": true,
        "lastFileUpdate": "Jan 8, 2020 11:07:44 PM",
        "primaryDirector": null,
        "secondaryDirector": null,
        "configuration": "default"
}
```


default-properties

Retrieves the names and descriptions from the default Engine property list.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/default-
properties
```

Example Response


```
[
    {
        "name": "Location",
        "value": "machine location"
    },
    {
        "name": "Group",
        "value": "work group to attach engine"
    },
    {
        "name": "QuarantineStatus",
        "value": "quarantine status of the engine"
    },
    {
        "name": "Description",
        "value": "brief description of machine"
    }
]
```


engine-daemon-count

Retrieves the total number of Engine Daemons logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/engine-daemon-count
```

Example Response

`1`.

Result: The count of Engine Daemons logged in to the Director is returned.

engine-daemon-ids

Retrieves the list of Engine Daemon IDs logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/engine-daemon-ids
```

Example Response

```
[

    4848394891525229394

]
```

engine-daemon-info

Retrieves information about the given Engine Daemon logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/engine-daemon-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
id	Long	Engine daemon ID of the Engine Daemon whose information needs to be retrieved	6153431282294490153

Example Response


```
"{
    "properties": [
        {
            "name": "ClientPlatform",
            "value": "win64"
        },
        {
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "availableDiskInMB",
            "value": "13788"
        },
        {
            "name": "IP",
```

```
            "value": "10.128.88.142"
        },
        {
            "name": "numInstances",
            "value": "0"
        },
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "username",
            "value": "win64vm142"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.142:27159/work"
        },
        {
            "name": "freeMemInKB",
            "value": "1324600"
        },
```

```
        {
            "name": "ClientEngineDaemonVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "os",
            "value": "win64"
        },
        {
            "name": "timeStamp",
            "value": "1579081956825"
        },
        {
            "name": "cpuUtilization",
            "value": "1"
        },
        {
            "name": "cpuSocketCount",
```

```
            "value": "1"
        },
        {
            "name": "id",
            "value": "6153431282294490153"
        },
        {
            "name": "freeDiskInMB",
            "value": "13788"
        },
        {
            "name": "dsCpuUtilization",
            "value": "1"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "guid",
            "value": "deadbeef8142"
        },
```

```
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "Timestamp",
            "value": "1579005517350"
        },
        {
            "name": "osName",
            "value": "WIN2K12"
        },
        {
            "name": "cpuTotal",
            "value": "2704.3"
        },
        {
            "name": "instance",
```

```
            "value": "0"
        },
        {
            "name": "osUsername",
            "value": "SYSTEM"
        }
    ],
    "enabled": true,
    "lastFileUpdate": "N/A",
    "primaryDirector": null,
    "secondaryDirector": null,
    "configuration": "default",
    "instances": -2,
    "engineId": 6153431282294490153,
    "installDate": "2020-01-14 00:31:17.682",
    "lastLoginDate": "Jan 14, 2020 4:38:37 AM",
    "autoStart": false,
    "manualStart": true
}"
```


engine-daemon-info-by-properties

Retrieves information about all the Engine Daemons logged in to the Director that match the Engine Condition.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/engine-daemon-info-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine Daemon condition data based on which the Engines need to be retrieved	{ "sets": null, "conditions": [{ "name": "IP", "comparison": 1, "value": "10.128 "nullCompare": f }, { "name": "os", "comparison": 1, "value": "win64" "nullCompare": f }],

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			"type": 0 }

Example Response


```
[
    {
        "properties": [
            {
                "name": "ClientPlatform",
                "value": "win64"
            },
            {
                "name": "cpuCoreCount",
                "value": "1"
            },
            {
                "name": "availableDiskInMB",
                "value": "13792"
            },
            {
```

```
            "name": "IP",
            "value": "10.128.88.142"
        },
        {
            "name": "numInstances",
            "value": "1"
        },
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "username",
            "value": "win64vm142"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.142:27159/work"
        },
        {
            "name": "freeMemInKB",
            "value": "1263052"
```

```
        },
        {
            "name": "ClientEngineDaemonVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "os",
            "value": "win64"
        },
        {
            "name": "timeStamp",
            "value": "1578998880502"
        },
        {
            "name": "cpuUtilization",
            "value": "1"
        },
        {
```

```
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "id",
            "value": "6153431282294490153"
        },
        {
            "name": "freeDiskInMB",
            "value": "13792"
        },
        {
            "name": "dsCpuUtilization",
            "value": "1"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "guid",
            "value": "deadbeef8142"
```

```
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "Timestamp",
            "value": "1578997789590"
        },
        {
            "name": "osName",
            "value": "WIN2K12"
        },
        {
            "name": "cpuTotal",
            "value": "2704.3"
        },
        {
```

```
            "name": "instance",
            "value": "0"
        },
        {
            "name": "osUsername",
            "value": "SYSTEM"
        }
    ],
    "enabled": true,
    "lastFileUpdate": "N/A",
    "primaryDirector": null,
    "secondaryDirector": null,
    "configuration": "default",
    "instances": -2,
    "engineId": 6153431282294490153,
    "installDate": "2020-01-14 00:31:17.682",
    "lastLoginDate": "Jan 14, 2020 2:29:49 AM",
    "autoStart": false,
    "manualStart": true
},
{
    "properties": [
```

```
        {
            "name": "ClientPlatform",
            "value": "win64"
        },
        {
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "availableDiskInMB",
            "value": "14090"
        },
        {
            "name": "IP",
            "value": "10.128.88.108"
        },
        {
            "name": "numInstances",
            "value": "1"
        },
        {
            "name": "QuarantineStatus",
```

```
            "value": "New Engine"
        },
        {
            "name": "username",
            "value": "win64vm108"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.108:27159/work"
        },
        {
            "name": "freeMemInKB",
            "value": "7412"
        },
        {
            "name": "ClientEngineDaemonVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
```

```
        {
            "name": "os",
            "value": "win64"
        },
        {
            "name": "timeStamp",
            "value": "1578998850072"
        },
        {
            "name": "cpuUtilization",
            "value": "7"
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "id",
            "value": "9007929106493580894"
        },
        {
            "name": "freeDiskInMB",
```

```
            "value": "14090"
        },
        {
            "name": "dsCpuUtilization",
            "value": "0"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "guid",
            "value": "deadbeef8108"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
```

```
            {
                "name": "Timestamp",
                "value": "1578974352009"
            },
            {
                "name": "osName",
                "value": "WIN2K12"
            },
            {
                "name": "cpuTotal",
                "value": "2794.4"
            },
            {
                "name": "instance",
                "value": "0"
            },
            {
                "name": "osUsername",
                "value": "SYSTEM"
            }
        ],
        "enabled": true,
```

```
        "lastFileUpdate": "N/A",
        "primaryDirector": null,
        "secondaryDirector": null,
        "configuration": "default",
        "instances": -2,
        "engineId": 9007929106493580894,
        "installDate": "2020-01-12 22:55:40.340",
        "lastLoginDate": "Jan 13, 2020 7:59:12 PM",
        "autoStart": false,
        "manualStart": true
    }
]
```


log-url-list

Retrieves the log file URLs from the given Engine Daemon when Engine Logging is enabled on the Server.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/log-url-list
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID for which the log file URLs need to be retrieved	`4848394891525229394`

Example Response

```
[

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.08.2020-23.08.11.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.54.36.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.54.18.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-0/log/
engine-0.01.06.2020-06.53.42.log",

    "http://10.128.88.91:27159/work/win64vm091-1/log/
engine-1.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-2/log/
engine-2.01.08.2020-22.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-3/log/
engine-3.01.08.2020-22.53.52.log",

    "http://10.128.88.91:27159/work/win64vm091-4/log/
engine-4.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-5/log/
engine-5.01.08.2020-22.53.52.log",
```

```
    "http://10.128.88.91:27159/work/win64vm091-6/log/
engine-6.01.08.2020-22.54.02.log",

    "http://10.128.88.91:27159/work/win64vm091-7/log/
engine-7.01.08.2020-22.54.01.log",

    "http://10.128.88.91:27159/work/win64vm091-8/log/
engine-8.01.08.2020-22.53.55.log",

    "http://10.128.88.91:27159/work/win64vm091-9/log/
engine-9.01.08.2020-22.53.52.log"

]
```

selected-engine-daemon-info

Retrieves information about a given Engine Daemon logged in to the Director.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/selected-
engine-daemon-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of Engine Daemon IDs whose information needs to be retrieved	[6153431282294490153, 9007929106493580894]

Example response


```
[
    {
        "properties": [
            {
                "name": "ClientPlatform",
                "value": "win64"
            },
            {
                "name": "cpuCoreCount",
                "value": "1"
            },
            {
                "name": "availableDiskInMB",
                "value": "13793"
            },
            {
                "name": "IP",
                "value": "10.128.88.142"
            },
            {
                "name": "numInstances",
```

```
            "value": "0"
        },
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "username",
            "value": "win64vm142"
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.142:27159/work"
        },
        {
            "name": "freeMemInKB",
            "value": "1355704"
        },
        {
            "name": "ClientEngineDaemonVersion",
            "value": "7.1.1.xxxxxx"
        },
```

```
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "os",
            "value": "win64"
        },
        {
            "name": "timeStamp",
            "value": "1578999181503"
        },
        {
            "name": "cpuUtilization",
            "value": "0"
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "id",
```

```
            "value": "6153431282294490153"
        },
        {
            "name": "freeDiskInMB",
            "value": "13793"
        },
        {
            "name": "dsCpuUtilization",
            "value": "0"
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "guid",
            "value": "deadbeef8142"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
```

```
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "Timestamp",
            "value": "1578997789590"
        },
        {
            "name": "osName",
            "value": "WIN2K12"
        },
        {
            "name": "cpuTotal",
            "value": "2704.3"
        },
        {
            "name": "instance",
            "value": "0"
        },
        {
            "name": "osUsername",
```

```
            "value": "SYSTEM"
        }
    ],
    "enabled": true,
    "lastFileUpdate": "N/A",
    "primaryDirector": null,
    "secondaryDirector": null,
    "configuration": "default",
    "instances": -2,
    "engineId": 6153431282294490153,
    "installDate": "2020-01-14 00:31:17.682",
    "lastLoginDate": "Jan 14, 2020 2:29:49 AM",
    "autoStart": false,
    "manualStart": true
},
{
    "properties": [
        {
            "name": "ClientPlatform",
            "value": "win64"
        },
        {
```

```
            "name": "cpuCoreCount",
            "value": "1"
        },
        {
            "name": "availableDiskInMB",
            "value": "14090"
        },
        {
            "name": "IP",
            "value": "10.128.88.108"
        },
        {
            "name": "numInstances",
            "value": "1"
        },
        {
            "name": "QuarantineStatus",
            "value": "New Engine"
        },
        {
            "name": "username",
            "value": "win64vm108"
```

```
        },
        {
            "name": "workUrl",
            "value": "http://10.128.88.108:27159/work"
        },
        {
            "name": "freeMemInKB",
            "value": "53068"
        },
        {
            "name": "ClientEngineDaemonVersion",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "Version",
            "value": "7.1.1.xxxxxx"
        },
        {
            "name": "os",
            "value": "win64"
        },
        {
```

```
            "name": "timeStamp",
            "value": "1578999461332"
        },
        {
            "name": "cpuUtilization",
            "value": "6"
        },
        {
            "name": "cpuSocketCount",
            "value": "1"
        },
        {
            "name": "id",
            "value": "9007929106493580894"
        },
        {
            "name": "freeDiskInMB",
            "value": "14090"
        },
        {
            "name": "dsCpuUtilization",
            "value": "0"
```

```
        },
        {
            "name": "cpuNo",
            "value": "1"
        },
        {
            "name": "guid",
            "value": "deadbeef8108"
        },
        {
            "name": "totalMemInKB",
            "value": "2096696"
        },
        {
            "name": "cpuThreadCount",
            "value": "1"
        },
        {
            "name": "Timestamp",
            "value": "1578974352009"
        },
        {
```

```
            "name": "osName",
            "value": "WIN2K12"
        },
        {
            "name": "cpuTotal",
            "value": "2794.4"
        },
        {
            "name": "instance",
            "value": "0"
        },
        {
            "name": "osUsername",
            "value": "SYSTEM"
        }
    ],
    "enabled": true,
    "lastFileUpdate": "N/A",
    "primaryDirector": null,
    "secondaryDirector": null,
    "configuration": "default",
    "instances": -2,
```

```
        "engineId": 9007929106493580894,
        "installDate": "2020-01-12 22:55:40.340",
        "lastLoginDate": "Jan 13, 2020 7:59:12 PM",
        "autoStart": false,
        "manualStart": true
    }
]
```

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/engineDaemon/available
```

Example Response

`True` or `False`

default-property

Removes the default property.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/engineDaemon/default-
property
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`property`	String	Default property that needs to be removed	`NewProp`

Example Response

```
new property description
```

Result: The description of the removed property is returned.

property

Removes a property from the Engine Daemon.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/engineDaemon/property
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID whose property needs to be removed	`6153431282294490153`
`key`	String	Key of the property that needs to be removed	`Location`

Example Response

```
machine location
```

Result: The description of the removed property is returned.

property-by-properties

Removes a property from selected Engine Daemons that match the Engine Condition.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/engineDaemon/property-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine Condition data depending on which engines are selected	{ "sets": null, "conditions": [{ "name": "osName" "comparison": 1, "value": "RHEL6" "nullCompare": f }], "type": 0 }

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`key`	String	Key of the property that needs to be removed	`NewProp`

Example Response

`1`

Result: Returns 1 if there is no exception.

restart-engine-daemon

Forces the Engine Daemon to log off and restart.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/restart-engine-daemon
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID of Engine Daemon that is forced to log off and restart	`4848394891525229394`

Example Response

`204 no content`

Result: Engine Daemon is restarted.

restart-engine-daemon-by-properties

Forces those Engine Daemons that match the Engine Condition to log off and restart.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/restart-engine-daemon-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Engine Condition data depending on which engines need to be selected for log off	<pre>{
			"sets": null,
			"conditions": [
			{
			"name": "osName"
			"comparison": 1,
			"value": "RHEL6"
			"nullCompare": f
			}
			],
			"type": 0
			}</pre>

Example Response

`204 no content`

Result: The Engine Daemon matching the specified condition is restarted.

all-enabled

Sets whether all Engine Daemons can run Engines.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/all-enabled
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`enable`	Boolean	True or false depending on which it is decided whether all Engine Daemons can run the Engines or not	`true`

Example Response

`204 no content`

Result: All Engine Daemons are enabled or disabled.

all-start-mode

Sets whether all Engines have to be started manually or automatically.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/all-start-mode
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`start`	Boolean	True or false. True states manual start and false states automatic start.	`true`

Example Response

`204 no content`

Result: The manual start property of all Engine Daemons is set to true or false, depending on what you have specified.

configuration

Sets the configuration to run on the Engines.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/configuration
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine ID for which configuration needs to be changed	`3574664343712327623`
`config`	String	The configuration which needs to be updated	`testlin64`

Example Response

`204 no content`

Result: Configuration of the specified Engine Daemon is updated.

configuration-by-properties

Sets the configuration to run on the Engines that match the Engine Condition.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/configuration-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine condition data depending on which engines need to be selected	{ "sets": null, "conditions": [{ "name": "os", "comparison": 1, "value": "linux6 "nullCompare": f }], "type": 0 }

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`config`	String	Configuration which needs to be updated	`testlin64`

Example Response

`204 no content`

Result: Configuration of the Engine Daemons matching the given condition is updated.

default-property

Creates a new default Engine Property and its description.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/default-property
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`property`	String	Name of the new default Engine property that needs to be created	`NewProp`
`description`	String	Description on the new default Engine property	`New Property description`

Example Response

`204 no content`

Result:A new default property is added.

directors

Sets the primary and secondary Directors for the Daemons, overriding the values in the configuration.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/directors
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID whose Primary and Secondary Directors whose primary and secondary Directors need to be overridden	`3574664343712327623`
`primary`	String	Name of the Primary Director	`http://example.com:8080`
`secondary`	String	Name of Secondary Director	`http://win64vm218.rofa.tibco.com:8080`

Example Response

`204 no content`

Result: The primary and secondary Directors for the specified Engine Daemon are set.

directors-by-properties

Sets the primary and secondary Directors on the Daemons for the Daemons that match the Engine condition, overriding the values in the configuration.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/directors-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine condition data depending on which Engines you select	`http://example.com:8080`
`primary`	String	Name of the Primary Director	`http://example.com:8080`
`secondary`	String	Name of the Secondary Director	`http://example.com:8080`

Example Response

`204 no content`

Result: The primary and secondary Directors are set for the Engine Daemons matching the given condition.

enabled

Sets whether the Engine Daemon can run Engines.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/enabled
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
id	Long	Engine Daemon ID	6153431282294490153
enable	Boolean	Engine Daemon enabled flag	False

Example Response

```
204 no content
```

Result: The specified Engine Daemon is enabled or disabled.

enabled-by-properties

Sets whether the Engine Daemon that matches the Engine Condition can run Engines.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/enabled-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Engine condition data depending on	{ "sets": null,

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
		which engines are selected	"conditions": [{ "name": "os", "comparison": 1, "value": "win64" "nullCompare": f }, { "name": "id", "comparison": 1, "value": "900792 "nullCompare": f }], "type": 0 }
enable	Boolean	Engine Daemon enabled flag. True or false.	true

Example Response

`204 no content`

Result: The Engine Daemons matching condition are enabled or disabled.

instances

Sets the number of Engine instances to run, overriding the value in the configuration.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/instances
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID for which the number of instances need to be updated	`6153431282294490153`
`instances`	Int	Number of instances that need to be overridden	`2`

Example Response

`204 no content`

Result: The number of Engine instances to run for Engine Daemons is set.

instances-by-properties

Sets the number of Engine instances to run, overriding the value in the configuration for Engines that match the Engine Conditions.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/instances-by-
properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Engine Condition data depending on which Engines need to be selected	<pre>{
			"sets": null,
			"conditions": [
			{
			"name": "os",
			"comparison": 1,
			"value": "linux6
			"nullCompare": f
			}
			],
			"type": 0
			}</pre>

`instances`	Int	Number of instances which need to be overridden	`3`

Example Response

`204 no content`

Result: The number of Engine instances to run is set for Engine Daemons matching the condition.

property

Sets a property on the Engine Daemon.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/property
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID for which a new property needs to be set	`9007929106493580894`
`key`	String	Key of the new property that needs to be added	`instances`
`value`	String	Value of the new property that needs to be added	`4`

Example Response

`204 no content`

Result:A new property of the Engine Daemon is set.

property-by-properties

Sets a property on selected Engine Daemons that match the Engine Condition.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/property-by-
properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Engine Condition data depending on which engines need to be selected	<pre>{
			"sets": null,
			"conditions": [
			{
			"name": "osName"
			"comparison": 1,
			"value": "RHEL6"
			"nullCompare": f
			}
			],
			"type": 0
			}</pre>

`key`	String	Key of the new property that needs to be added	`New Prop`
`value`	String	Value of the new property that needs to be added	`New Prop Value`

Example Response

`204 no content`

Result:A property is set for the selected Engine Daemons that match the Engine condition.

start-mode

Sets whether the Engines have to be started manually or automatically.

Example Request

```
PUT http://example.com:8080/livecluster/rest/engineDaemon/start-mode
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`id`	Long	Engine Daemon ID for which start mode needs to be set	`9007929106493580894`
`start`	Boolean	True or False. True states manual start and false states automatic start.	`true`

Example Response

`204 no content`

Result: The property of all the specified Engine Daemons to start manually is set to true or false.

start-mode-by-properties

Sets whether the Engines that match the Engine Condition have to be started manually or automatically.

Example Request

```
POST http://example.com:8080/livecluster/rest/engineDaemon/start-mode-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Engine Condition data depending on which engines need to be selected	<pre>{
			"sets": null,
			"conditions": [
			{
			"name": "os",
			"comparison": 1,
			"value": "linux6
			"nullCompare": f
			}
			],
			"type": 0
			}</pre>

`manual`	Boolean	Manual startup flag. True or false.	`true`

Example Response

`204 no content`

Result: The property to start manually for Engine Daemons that match the condition is set to true or false.

DriverManager

DirverManager API provides a method to retrieve the base context URL. The method is listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
broker-url	GET	Retrieves the base context URL of the Brokers

broker-url

Retrieves the base context URL of the Brokers

Example Request

```
GET http://example.com:8080/livecluster/rest/driverManager/broker-url
```

Example Response

http://example.com:8080/livecluster/.

Result: All the base context URLs of the Brokers are returned.

ManagerAdmin

Provides administrative access to the Manager. Methods in class ManagerAdmin are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
broker-id	GET	Retrieves the Broker ID if it exists else returns -1.
broker-name	GET	Retrieves the Broker name if it exists else returns null.
broker-url	GET	Retrieves the Broker URL; returns null if the Manager does not contain a Broker.

METHOD	METHOD TYPE	DESCRIPTION
build-version	GET	Retrieves the full Manager version including the build number.
busy-engine-count	GET	Retrieves the total number of busy Engines from all the Brokers logged in to the Director.
category	GET	Retrieves the specified configuration category.
category-names	GET	Retrieves the Manager configuration category names.
director-id	GET	Gets the Director ID if it exists else returns -1.
engine-configuration-names	GET	Retrieves a list of names of the Engine configuration profiles available on the specified Manager.
engine-count	GET	Retrieves the total number of Engines from all the Brokers logged in to the Director.
events	GET	Retrieves the available subscription events.
finished-service-count	GET	Retrieves the total number of finished Services from all the Brokers logged in to the Director.
license-info	GET	Retrieves the Manager license information.
manager-value	GET	Retrieves a Manager property value specified by category name, property group name,

METHOD	METHOD TYPE	DESCRIPTION
		and property name on the specified manager.
pending-invocation-count	GET	Retrieves the total number of pending Service invocations from all the Brokers logged in to the Director.
running-invocation-count	GET	Retrieves the total number of running Service invocations from all the Brokers logged in to the Director.
running-service-count	GET	Retrieves the total number of running Services from all the Brokers logged in to the Director.
service-count	GET	Retrieves the total number of Services from all the Brokers logged in to the Director.
subscriber-events	POST	Retrieves the events that the given subscriber is registered to receive.
subscribers	GET	Retrieves the registered event subscribers.
value	GET	Gets a Manager property value specified by the category name, property group name, and property name.
version	GET	Retrieves the Manager version in M.m format.
available	GET	Retrieves whether the methods are available.

METHOD	METHOD TYPE	DESCRIPTION
manager-value	PUT	Sets a Manager property value specified by the category name, property group name, and property name on the specified manager.
value	PUT	Sets a Manager property value specified by the category name, property group name, and property name.
subscribe	POST	Subscribes to the given events.
unsubscribe	POST	Unsubscribes from the given events.

broker-id

Retrieves the Broker ID if it exists else returns -1.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/broker-id
```

Example Response

```
899860077
```

broker-name

Retrieves the Broker name if it exists else returns null.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/broker-name
```

Example Response

`800860077`.

Result: The name of the Broker is returned.

broker-url

Retrieves the Broker URL; returns null if the Manager does not contain a Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/broker-url
```

Example Response

`http: //<hostname>:8000/livecluster`

build-version

Retrieves the full Manager version including the build number.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/build-version
```

Example Response

`7.1.1.xxxxxx`

busy-engine-count

Retrieves the total number of busy Engines from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/busy-engine-count
```

Example Response

`0`

category

Retrieves the specified configuration category.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/category
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
name	String	Name of the category depending on which configuration needs to be retrieved	Logging

Example Response

```
{
    "name": "Logging",
    "groups": [
        {
            "name": "General",
            "properties": [
                {
                    "name": "Default Debug Level",
                    "value": "INFO"
                },
                {
                    "name": "Use the Fully Qualified Class Name",
```

```
                "value": "false"
            },
            {
                "name": "Log Level List",
                "value": ""
            }
        ]
    },
    {
        "name": "File",
        "properties": [
            {
                "name": "Stack Trace",
                "value": "true"
            },
            {
                "name": "Log To File",
                "value": "true"
            },
            {
                "name": "Max File Length",
                "value": "1000000"
```

```
            },
            {
                "name": "Log File Directory",
                "value": "F:/datasynapse/manager-data/logs/server"
            },
            {
                "name": "Max Log Level",
                "value": "ALL"
            },
            {
                "name": "Time Zone",
                "value": ""
            }
        ]
    },
    {
        "name": "Log to REST",
        "properties": [
            {
                "name": "Server URL",
                "value": ""
            },
```

```
            {
                "name": "Enabled",
                "value": "false"
            },
            {
                "name": "Username",
                "value": ""
            },
            {
                "name": "Password",
                "value": null
            },
            {
                "name": "Max Log Level",
                "value": "ALL"
            },
            {
                "name": "Backlog",
                "value": "100"
            },
            {
                "name": "Discard when Backlogged",
```

```
                    "value": "true"
                },
                {
                    "name": "Timestamp format",
                    "value": "yyyy-MM-dd'T'HH:mm:ss.SSSZZ"
                },
                {
                    "name": "Excluded LogRecord Properties",
                    "value":
"millis,sequenceNumber,sourceClassName,sourceMethodName"
                },
                {
                    "name": "HTTP Timeout",
                    "value": "5000"
                }
            ]
        },
        {
            "name": "Server Events to REST",
            "properties": [
                {
                    "name": "Enabled",
```

```
            "value": "false"
        },
        {
            "name": "Server URL",
            "value": ""
        },
        {
            "name": "Username",
            "value": ""
        },
        {
            "name": "Password",
            "value": null
        },
        {
            "name": "Backlog",
            "value": "1000"
        },
        {
            "name": "Discard when Backlogged",
            "value": "false"
        },
```

```
            {
                "name": "Timestamp format",
                "value": "yyyy-MM-dd'T'HH:mm:ss.SSSZZ"
            },
            {
                "name": "Excluded Events",
                "value": ""
            },
            {
                "name": "HTTP Timeout",
                "value": "5000"
            },
            {
                "name": "Include name",
                "value": "false"
            },
            {
                "name": "Include type",
                "value": "true"
            }
        ]
    },
```

```
{
    "name": "Syslog",
    "properties": [
        {
            "name": "Enabled",
            "value": "false"
        },
        {
            "name": "Time Zone",
            "value": ""
        },
        {
            "name": "Syslog Server Address",
            "value": ""
        },
        {
            "name": "Syslog Server Port",
            "value": "514"
        },
        {
            "name": "Components",
            "value": "*"
```

```
            },
            {
                "name": "Max Log Level",
                "value": "ALL"
            },
            {
                "name": "Tag",
                "value": "GridServerManager"
            },
            {
                "name": "Facility Level",
                "value": "1"
            }
        ]
    },
    {
        "name": "Component-level Logging",
        "properties": [
            {
                "name": "Engine Events",
                "value": "DEFAULT"
            },
```

```
        {
            "name": "Driver Events",
            "value": "DEFAULT"
        },
        {
            "name": "Service Events",
            "value": "DEFAULT"
        },
        {
            "name": "Scheduler",
            "value": "DEFAULT"
        },
        {
            "name": "Engine Sharing",
            "value": "DEFAULT"
        },
        {
            "name": "GridCache",
            "value": "DEFAULT"
        },
        {
            "name": "Batch Events",
```

```
                "value": "DEFAULT"
            },
            {
                "name": "Web Services",
                "value": "DEFAULT"
            }
        ]
    },
    {
        "name": "Log File Cleaner",
        "properties": [
            {
                "name": "File Cleaner Timeout",
                "value": "0"
            },
            {
                "name": "Cleaner Cron Expression",
                "value": "0 50 3 * * ?"
            },
            {
                "name": "File Time To Live",
                "value": "120.0"
```

```
            },
            {
                "name": "File Cleaner Enabled",
                "value": "true"
            }
        ]
    },
    {
        "name": "Tomcat Log File Cleaner",
        "properties": [
            {
                "name": "File Cleaner Timeout",
                "value": "0"
            },
            {
                "name": "Cleaner Cron Expression",
                "value": "0 0 * * * ?"
            },
            {
                "name": "File Time To Live",
                "value": "120.0"
            },
```

```
            {
                "name": "File Cleaner Enabled",
                "value": "true"
            }
        ]
    },
    {
        "name": "Memory Logging",
        "properties": [
            {
                "name": "Log Period",
                "value": "1.0"
            },
            {
                "name": "Warning Threshold",
                "value": "80"
            },
            {
                "name": "Total Allocated Heap",
                "value": "192"
            },
            {
```

```
                    "name": "Free Heap",
                    "value": "52"
                },
                {
                    "name": "Maximum Heap",
                    "value": "1015"
                }
            ]
        }
    ]
}
```

category-names

Retrieves the Manager configuration category names.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/category-names
```

Example Response

```
[
    "Logging",
    "Services",
    "Engines and Clients",
    "Admin",
```

```
    "Database",
    "Resource Deployment",
    "Cache",
    "Security",
    "Communication",
    "Hotfixes"
]
```

director-id

Gets the Director ID if it exists else returns -1.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/director-id
```

Example Response

`899860077`

engine-configuration-names

Retrieves a list of names of the Engine configuration profiles available on the specified Manager.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/engine-configuration-
names
```

Example Response

```
[


    "linux64:default",
```

```
    "win64:default",
    "win32:default",
    "linux:default"
]
```

engine-count

Retrieves the total number of Engines from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/engine-count
```

Example Response

```
10
```

events

Retrieves the available subscription events.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/events
```

Example Response

```
[
    "ServiceEvent",
    "ServicePriorityChangedEvent",
    "EngineDiedEvent",
    "EngineAddedEvent",
    "EngineRemovedEvent",
```

```
    "EngineBlacklistedEvent",
    "EngineGreylistedEvent",
    "EngineDaemonAddedEvent",
    "EngineDaemonRemovedEvent",
    "DriverAddedEvent",
    "DriverRemovedEvent",
    "BrokerAddedEvent",
    "BrokerRemovedEvent",
    "LicenseEvent",
    "ServerInitializedEvent",
    "MemoryWarningEvent"
]
```

finished-service-count

Retrieves the total number of finished Services from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/finished-service-count
```

Example Response

```
3
```

license-info

Retrieves the Manager license information.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/license-info
```

Example Response

```
{

    "properties": [

        {

            "name": "licenseVersion",

            "value": "5.0"

        },

        {

            "name": "maxDaemons",

            "value": "632000"

        },

        {

            "name": "productname",

            "value": "GridServer"

        },

        {

            "name": "maxBrokers",

            "value": "632000"

        },

        {
```

```
            "name": "maxCpuThreadNo",
            "value": "632000"
        },
        {
            "name": "maxCpuCoreNo",
            "value": "632000"
        },
        {
            "name": "maxEngines",
            "value": "632000"
        },
        {
            "name": "maxCpuSocketNo",
            "value": "632000"
        },
        {
            "name": "maxCpuNo",
            "value": "632000"
        }
    ],
    "errorMessage": null,
    "customerName": null,
```

```
    "allowedHostnames": [
        "*:*"
    ],
    "expirationDate": 1787447752000,
    "featureSets": [
        {
            "name": "drivers",
            "value": "jdriver pdriver cppdriver NETdriver SOAPdriver
DWSdriver COMdriver"
        },
        {
            "name": "serviceTypes",
            "value": "*"
        },
        {
            "name": "schedulers",
            "value": "SLA"
        },
        {
            "name": "engineOS",
            "value": "*"
        },
```

```
        {
            "name": "serverOS",
            "value": "*"
        }
    ]
}
```

manager-value

Retrieves a Manager property value specified by category name, property group name, and property name on the specified manager.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/manager-value
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`manager`	String	Manger ID for which property needs to be retrieved	`1580108171`
`category`	String	Name of category	`Logging`
`propertyGroup`	String	Name of property group	`General`
`name`	String	Name of property	`Default debug level`

Example Response

```
INFO
```

Result: The corresponding value of the property is returned.

pending-invocation-count

Retrieves the total number of pending Service invocations from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/pending-invocation-count
```

Example Response

```
0
```

running-invocation-count

Retrieves the total number of running Service invocations from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/running-invocation-count
```

Example Response

```
0
```

running-service-count

Retrieves the total number of running Services from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/running-service-count
```

Example Response

```
0
```

service-count

Retrieves the total number of Services from all the Brokers logged in to the Director.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/service-count
```

Example Response

```
2
```

subscriber-events

Retrieves the events that the given subscriber is registered to receive.

Example Request

```
POST http://example.com:8080/livecluster/rest/manager/subscriber-events
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Subscriber data depending on which events need to be retrieved	{ "filter": "pqr", "email": "test@tibco.com }

Example Response

```
[

    "ServiceEvent",

    "ServicePriorityChangedEvent"

]
```

subscribers

Retrieves the registered event subscribers.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/subscribers
```

Example Response

```
[

    {

        "filter": null,

        "email": "test"

    }

]
```

value

Gets a Manager property value specified by the category name, property group name, and property name.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/value
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`category`	String	Name of the category	`Logging`
`propertyGroup`	String	Name of the property group	`General`
`name`	String	Name of the property	`Default Debug Level`

Example Response

`INFO`

version

Retrieves the Manager version in M.m format.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/version
```

Example Response

`7.1`

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/manager/available
```

Example Response

`True` or `False`

manager-value

Sets a Manager property value specified by the category name, property group name, and property name on the specified manager.

Example Request

```
PUT http://example.com:8080/livecluster/rest/manager/manager-value
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`manager`	String	Manager ID for which property needs to be retrieved	`1580108171`
`category`	String	Name of the category	`Logging`
`propertyGroup`	String	Name of the property group	`General`
`name`	String	Name of the property	`Default Debug Level`
`value`	String	Value of the property	`FINE`

Example Response

`204 no content`

Result: Property value is updated correctly.

value

Sets a Manager property value specified by the category name, property group name, and property name.

Example Request

```
PUT http://example.com:8080/livecluster/rest/manager/value
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`category`	String	Name of the category	`Logging`
`propertyGroup`	String	Name of the property group	`General`
`name`	String	Name of the property	`Default Debug Level`
`value`	String	Value of the category	`FINE`

Example Response

`204 no content`

Result: Property value is updated correctly.

subscribe

Subscribes to the given events.

Example Request

```
POST http://example.com:8080/livecluster/rest/manager/subscribe
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Subscriber data that subscribes to given events	<pre>{
			"subscriber": {
			"filter": "xyz",
			"email": "test@examp
			},
			"events": [
			"ServiceEvent",
			"ServicePriorityChan
			]
			}</pre>

Example Response

`204 no content`

Result: Subscribed to given event.

unsubscribe

Unsubscribes from the given events.

Example Request

```
POST http://example.com:8080/livecluster/rest/manager/unsubscribe
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Subscriber data that unsubscribes from given events	<pre>{
			"subscriber": {
			"filter": "xyz",
			"email": "test@examp
			},
			"events": [
			"ServiceEvent",
			"ServicePriorityChan
			]
			}</pre>

Example Response

`204 no content`

Result: Unsubscribed from given event.

ServiceAdmin

ServiceAdmin APIs provide administrative access to the Services on a Broker. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
all-services	PUT	Cancels all active Services on the Broker.

METHOD	METHOD TYPE	DESCRIPTION
invocation	PUT	Cancels a specific invocation specified by the given Service ID and Invocation ID.
service	PUT	Cancels the Service specified by the given Service ID.
resources	DELETE	Removes resource files from both the staging and deployed areas.
deploy-resources	POST	Deploys resources.
all-service-info	GET	Retrieves Service information of all the Services on the Broker.
blacklisted-engines	GET	Retrieves a list of IDs of Engines that have been blacklisted for a specified Service.
completed-service-invocation-count	GET	Retrieves the number of completed invocations of the specified Service.
finished-service-count	GET	Retrieves the total number of finished Services on the Broker.
invocation-count	GET	Retrieves the total number of pending and running invocations on the Broker.
invocation-info	GET	Retrieves the information about a particular invocation of a Service, for the given Service ID and Invocation ID.

METHOD	METHOD TYPE	DESCRIPTION
pending-invocation-count	GET	Retrieves the number of pending invocations on the Broker.
pending-service-invocation-count	GET	Retrieves the number of pending invocations of the specified Service.
registered-services	GET	Retrieves a list of all registered Services; returns null if no Services are registered.
running-invocation-count	GET	Retrieves the number of invocations currently executed by Engines on the Broker.
running-service-count	GET	Retrieves the total number of active Services on the Broker.
running-service-invocation-count	GET	Retrieves the number of invocations currently executed by Engines of the specified Service.
selected-invocation-info	POST	Retrieves information of the given Service and invocation IDs.
selected-service-info	POST	Retrieves information of the given Service IDs running on the Broker.
service-binding	GET	Retrieves the binding of the specified Service.
service-count	GET	Retrieves the total number of Services on the Broker.

METHOD	METHOD TYPE	DESCRIPTION
service-ids	GET	Retrieves the list of all Service IDs on the Broker.
service-info	GET	Retrieves the Service information of the given Service ID.
service-info-by-properties	POST	Retrieves the Service information of all Services on the Broker that match the description condition.
service-invocation-count	GET	Retrieves the total number of invocations of the specified Service.
task-expiration-event-count	GET	Retrieves the total number of task expiration events of the specified Service.
available	GET	Retrieves whether the methods are available.
list-resources	GET	Lists resources on the Server.
register-service	POST	Registers the binding in the Service Registry; this must be done on the Primary Director.
all-finished-services	DELETE	Removes the information of all finished Services from the Broker.
finished-service	DELETE	Removes the information of finished Services from the Broker.
resource-exists	GET	Checks whether a file exists in the designated area.

METHOD	METHOD TYPE	DESCRIPTION
expires	PUT	Sets whether Service information is removed during Service cleanup.
priority	PUT	Sets the priority of the specified Service.
unregister-service	DELETE	Unregisters the binding in the Service Registry; this must be done on the Primary Director.

all-services

Cancels all active Services on the Broker.

Example Request

```
PUT http://example.com:8080/livecluster/rest/service/all-services
```

Example Response

`204 no content`

Result: All active services on the broker are canceled.

invocation

Cancels a specific invocation specified by the given Service ID and Invocation ID.

Example Request

```
PUT http://example.com:8080/livecluster/rest/service/invocation
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Service ID of the Service that needs to be canceled	`4697058754240048269`
`taskId`	Long	Invocation ID of the Service that needs to be canceled	`5`

Example Response

`204 no content`

Result: Specific invocation for given service ID and Invocation ID is canceled.

service

Cancels the Service specified by the given Service ID.

Example Request

```
PUT http://example.com:8080/livecluster/rest/service/service
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Service ID of the Service that needs to be canceled	`6344484810359622656`

Example Response

`204 no content`

Result: Service specified by the given Service ID is canceled.

deploy-resources

Deploys resources.

Example Request

```
POST http://example.com:8080/livecluster/rest/service/deploy-resources
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Array of resource names that need to be deployed	`["calculator-1.0.0.1.tar.gz"`
`verifyGridLibraryStructure`	Boolean	Grid library structure flag. True or false.	`True or False`
`verifyGridLibraryDependencies`	Boolean	Grid library dependency flag. True or false.	`True or False`
`verifyGridLibraryNaming`	Boolean	Grid library naming flag. True or false.	`True or False`

Example Response

`204 no content`

Result: Corresponding resources are deployed.

all-service-info

Retrieves Service information of all Services on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/all-service-info
```

Example Response


```
[
    {
        "finishTime": -1,
        "pendingCount": 2,
        "runningCount": 8,
        "completedCount": 0,
        "totalCount": 10,
        "serviceName": "Linpack Test",
        "cancelled": false,
        "description": [
            {
                "name": "appName",
                "value": "DataSynapse Job/Service Test"
            },
            {
                "name": "serviceTypeName",
                "value": "LinpackServiceTest"
            },
            {
```

```
            "name": "serviceName",
            "value": "Linpack Test"
        },
        {
            "name": "class",
            "value": "java:examples.linpack.LinpackService"
        },
        {
            "name": "serviceType",
            "value": "Service"
        },
        {
            "name": "slaGroupName",
            "value": ""
        },
        {
            "name": "providerType",
            "value": "JavaProvider"
        },
        {
            "name": "username",
            "value": "internal"
```

```
            }
        ],
        "finished": false,
        "status": "Running",
        "options": [
            {
                "name": "autoPackMode",
                "value": "0"
            },
            {
                "name": "compressData",
                "value": "true"
            },
            {
                "name": "resultsPerMessage",
                "value": "100"
            },
            {
                "name": "resubmitOnDDTFailure",
                "value": "true"
            },
            {
```

```
            "name": "statusExpires",
            "value": "true"
        },
        {
            "name": "tasksPerMessage",
            "value": "20"
        },
        {
            "name": "deleteInvocationData",
            "value": "2"
        },
        {
            "name": "maxEngines",
            "value": "2147483647"
        },
        {
            "name": "taskMaxTime",
            "value": "9223372036854775807"
        },
        {
            "name": "priority",
            "value": "5"
```

```
        },
        {
            "name": "killCancelledTasks",
            "value": "true"
        },
        {
            "name": "checkpoint",
            "value": "false"
        },
        {
            "name": "unloadNativeLibrary",
            "value": "true"
        },
        {
            "name": "autoCancel",
            "value": "1"
        },
        {
            "name": "autoPackNum",
            "value": "0"
        },
        {
```

```
                "name": "collectionType",
                "value": "1"
            }
        ],
        "invocationInfo": [],
        "submitTime": 1643633925014,
        "driverHostname": "TIBCO-PF379VJL",
        "driverUsername": "internal",
        "resultProperties": null,
        "serviceId": 565597320224386901,
        "completed": false
    }
]
```


Result: Information about all Services is returned.

blacklisted-engines

Retrieves a list of IDs of Engines that have been blacklisted for a specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/blacklisted-engines
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`serviceId`	Long	Service ID for which black-listed Engine IDs need to be retrieved	`7655032223441706693`

Example Response

```
[

    4848394891525229394

]
```

Result:IDs of blacklisted Engines are returned.

completed-service-invocation-count

Retrieves the number of completed invocations of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/completed-service-invocation-count
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`serviceId`	Long	Service ID for which completed Service invocation count needs to be retrieved	`7655032223441706693`

Example response

```
0
```

finished-service-count

Retrieves the total number of finished Services on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/finished-service-count
```

Example Response

```
1
```

invocation-count

Retrieves the total number of pending and running invocations on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/invocation-count
```

Example Response

```
0
```

invocation-info

Retrieves the information about a particular invocation of a Service, for the given Service ID and Invocation ID.

Example Request

```
Get http://example.com:8080/livecluster/rest/service/invocation-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Service ID of the Service whose information needs to be retrieved	`4697058754240048269`
`taskId`	Long	Invocation ID of the Service whose information needs to be retrieved	`5`

Example Response


```
{
    "description": null,
    "engine": null,
    "startTime": 0,
    "elapsedTime": 0,
    "submitTime": 1578912618068,
    "exception": 0,
    "id": 5,
    "info": null,
    "status": "Submitted",
    "computationalTime": 0,
    "inputSize": 533,
    "outputSize": 0,
```

```
    "reschedules": 0,

    "retries": 0

}
```

pending-invocation-count

Retrieves the number of pending invocations on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/pending-invocation-count
```

Example Response

```
0
```

pending-service-invocation-count

Retrieves the number of pending invocations of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/pending-service-invocation-count
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`serviceId`	Long	Service ID for which Service invocation count needs to be retrieved	`7655032223441706693`

Example Response

```
0
```

registered-services

Retrieves a list of all registered Services; returns null if no Services are registered.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/registered-services
```

Example Response

```
[

    "CPPCalculatorExample",

    "CUDAExample",

    "JavaAdderExample",

    "JavaCalculatorExample",

    "MICExample",

    "NETCalculatorExample",

    "NETSpeedLink",

    "PythonExample",

    "RadhocExample",

    "RCalculatorExample",

    "RPICalculatorExample",

    "SpeedLink"

]
```

running-invocation-count

Retrieves the number of invocations currently executed by Engines on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/running-invocation-count
```

Example Response

```
0
```

running-service-count

Retrieves the total number of active Services on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/running-service-count
```

Example Response

```
0
```

running-service-invocation-count

Retrieves the number of invocations currently executed by Engines of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/running-service-invocation-count
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
serviceId	Long	Service ID for which running Service	7655032223441706693

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
		invocation count needs to be retrieved	

Example Response

```
10
```

selected-invocation-info

Retrieves information of the given Service and invocation IDs.

Example Request

```
POST http://example.com:8080/livecluster/rest/service/selected-invocation-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`serviceId`	Long	Service ID for which information needs to be retrieved	`4625254489192920053`
—	JSON	Invocation IDs for which information needs to be retrieved	`[1,5]`

Example Response

```
[

    {
```

```
        "description": null,
        "engine": {
            "instance": "0",
            "properties": null,
            "username": "win64vm108",
            "engineId": 9007929106493580894,
            "serviceId": -1,
            "invocationId": -1,
            "busy": false,
            "elapsedTime": -1
        },
        "startTime": 1578911378974,
        "elapsedTime": 10131,
        "submitTime": 1578911368704,
        "exception": 0,
        "id": 1,
        "info": "2,572.13 MFlops",
        "status": "Completed",
        "computationalTime": 10033,
        "inputSize": 533,
        "outputSize": 328,
        "reschedules": 0,
```

```
        "retries": 0
    },
    {
        "description": null,
        "engine": {
            "instance": "0",
            "properties": null,
            "username": "win64vm108",
            "engineId": 9007929106493580894,
            "serviceId": -1,
            "invocationId": -1,
            "busy": false,
            "elapsedTime": -1
        },
        "startTime": 1578911419455,
        "elapsedTime": 10082,
        "submitTime": 1578911368799,
        "exception": 0,
        "id": 5,
        "info": "2,180.66 MFlops",
        "status": "Completed",
        "computationalTime": 10065,
```

```
        "inputSize": 533,
        "outputSize": 328,
        "reschedules": 0,
        "retries": 0
    }
]
```

selected-service-info

Retrieves information of the given Service IDs running on the Broker.

Example Request

```
POST http://example.com:8080/livecluster/rest/service/selected-service-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	List of Service IDs for which information needs to be retrieved	`8640091942181654218`

Example Response

```
[
    {
        "finishTime": -1,
```

```
"pendingCount": 5,

"runningCount": 0,

"completedCount": 0,

"totalCount": 5,

"cancelled": false,

"serviceName": "Linpack Test",

"description": [

    {

        "name": "appName",

        "value": "DataSynapse Job/Service Test"

    },

    {

        "name": "serviceTypeName",

        "value": "LinpackServiceTest"

    },

    {

        "name": "serviceName",

        "value": "Linpack Test"

    },

    {

        "name": "class",

        "value": "java:examples.linpack.LinpackService"
```

```
            },
            {
                "name": "serviceType",
                "value": "Service"
            },
            {
                "name": "slaGroupName",
                "value": ""
            },
            {
                "name": "providerType",
                "value": "JavaProvider"
            },
            {
                "name": "username",
                "value": "internal"
            }
        ],
        "status": "Running",
        "options": [
            {
                "name": "autoPackMode",
```

```
            "value": "0"
        },
        {
            "name": "compressData",
            "value": "true"
        },
        {
            "name": "resultsPerMessage",
            "value": "100"
        },
        {
            "name": "resubmitOnDDTFailure",
            "value": "true"
        },
        {
            "name": "statusExpires",
            "value": "true"
        },
        {
            "name": "tasksPerMessage",
            "value": "20"
        },
```

```
        {
            "name": "deleteInvocationData",
            "value": "2"
        },
        {
            "name": "maxEngines",
            "value": "2147483647"
        },
        {
            "name": "taskMaxTime",
            "value": "9223372036854775807"
        },
        {
            "name": "priority",
            "value": "5"
        },
        {
            "name": "killCancelledTasks",
            "value": "true"
        },
        {
            "name": "checkpoint",
```

```
            "value": "false"
        },
        {
            "name": "unloadNativeLibrary",
            "value": "true"
        },
        {
            "name": "autoCancel",
            "value": "1"
        },
        {
            "name": "autoPackNum",
            "value": "0"
        },
        {
            "name": "collectionType",
            "value": "1"
        }
    ],
    "finished": false,
    "resultProperties": null,
    "invocationInfo": [],
```

```
        "completed": false,
        "submitTime": 1578909679845,
        "serviceId": 8640091942181654218,
        "driverHostname": "mkamdar-p50.apac.tibco.com",
        "driverUsername": "internal"
    }
]
```

service-binding

Retrieves the binding of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/service-binding
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
name	String	Name of the Service	LinpackServiceTest

Example Response

```
{
    "parameters": [
        {
            "name": "className",
```

```
            "value": "examples.linpack.LinpackService"
        }
    ],
    "options": null,
    "description": [
        {
            "name": "serviceTypeName",
            "value": "LinpackServiceTest"
        },
        {
            "name": "providerType",
            "value": "JavaProvider"
        }
    ],
    "name": "LinpackServiceTest",
    "uidescription": "",
    "type": "java",
    "containerBinding": [
        {
            "name": "serviceMethods",
            "value": "*"
        }
```

```
    ]

}
```

service-count

Retrieves the total number of Services on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/service-count
```

Example Response

```
1
```

service-ids

Retrieves the list of all Service IDs on the Broker.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/service-ids
```

Example Response

A list of Service IDs on the Broker is returned.

```
[

    4371703018904140926

]
```

service-info

Retrieves Service information of the given Service ID.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/service-info
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
jobId	Long	Service ID of the Service whose information needs to be retrieved	8656732499466570974

Example Response


```
{
    "finishTime": 1641820932754,
    "pendingCount": 0,
    "runningCount": 0,
    "completedCount": 10,
    "totalCount": 10,
    "serviceName": "Linpack Test",
    "cancelled": false,
    "description": [
        {
            "name": "appName",
            "value": "DataSynapse Job/Service Test"
        },
        {
```

```
            "name": "serviceTypeName",
            "value": "LinpackServiceTest"
        },
        {
            "name": "serviceName",
            "value": "Linpack Test"
        },
        {
            "name": "class",
            "value": "java:examples.linpack.LinpackService"
        },
        {
            "name": "serviceType",
            "value": "Service"
        },
        {
            "name": "slaGroupName",
            "value": ""
        },
        {
            "name": "providerType",
            "value": "JavaProvider"
```

```
        },
        {
            "name": "username",
            "value": "internal"
        }
    ],
    "status": "Finished",
    "finished": true,
    "options": [
        {
            "name": "autoPackMode",
            "value": "0"
        },
        {
            "name": "compressData",
            "value": "true"
        },
        {
            "name": "resultsPerMessage",
            "value": "100"
        },
        {
```

```
            "name": "resubmitOnDDTFailure",
            "value": "true"
        },
        {
            "name": "statusExpires",
            "value": "true"
        },
        {
            "name": "tasksPerMessage",
            "value": "20"
        },
        {
            "name": "deleteInvocationData",
            "value": "2"
        },
        {
            "name": "maxEngines",
            "value": "2147483647"
        },
        {
            "name": "taskMaxTime",
            "value": "9223372036854775807"
```

```
        },
        {
            "name": "priority",
            "value": "5"
        },
        {
            "name": "killCancelledTasks",
            "value": "true"
        },
        {
            "name": "checkpoint",
            "value": "false"
        },
        {
            "name": "unloadNativeLibrary",
            "value": "true"
        },
        {
            "name": "autoCancel",
            "value": "1"
        },
        {
```

```
            "name": "autoPackNum",
            "value": "0"
        },
        {
            "name": "collectionType",
            "value": "1"
        }
    ],
    "invocationInfo": [],
    "completed": true,
    "serviceId": 4371703018904140926,
    "resultProperties": null,
    "driverHostname": "TIBCO-PF379VJL",
    "driverUsername": "internal",
    "submitTime": 1641820912300
}
```


service-info-by-properties

Retrieves the Service information of all Services on the Broker that match the description condition.

Example Request

```
POST http://example.com:8080/livecluster/rest/service/service-info-by-properties
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE

—	JSON	Description condition data depending on information of which Services need to be retrieved	{
			"sets": null,
			"conditions": [
			{
			"name": "service
			"comparison": 1,
			"value": "Servic
			"nullCompare": f
			},
			{
			"name": "service
			"comparison": 1,
			"value": "Linpac
			"nullCompare": f
			}
			],
			"type": 0
			}

Example Response


```
"[
    {
        "finishTime": 1578980454833,
        "pendingCount": 0,
        "runningCount": 0,
        "completedCount": 100,
        "totalCount": 100,
        "cancelled": false,
        "serviceName": "Linpack Test",
        "description": [
            {
                "name": "appName",
                "value": "DataSynapse Job/Service Test"
            },
            {
                "name": "serviceTypeName",
                "value": "LinpackServiceTest"
            },
            {
                "name": "serviceName",
                "value": "Linpack Test"
```

```
            },
            {
                "name": "class",
                "value": "java:examples.linpack.LinpackService"
            },
            {
                "name": "serviceType",
                "value": "Service"
            },
            {
                "name": "slaGroupName",
                "value": ""
            },
            {
                "name": "providerType",
                "value": "JavaProvider"
            },
            {
                "name": "username",
                "value": "internal"
            }
        ],
```

```
    "status": "Finished, Task Errors",
    "options": [
        {
            "name": "autoPackMode",
            "value": "0"
        },
        {
            "name": "compressData",
            "value": "true"
        },
        {
            "name": "resultsPerMessage",
            "value": "100"
        },
        {
            "name": "resubmitOnDDTFailure",
            "value": "true"
        },
        {
            "name": "statusExpires",
            "value": "true"
        },
```

```
        {
            "name": "tasksPerMessage",
            "value": "20"
        },
        {
            "name": "deleteInvocationData",
            "value": "2"
        },
        {
            "name": "maxEngines",
            "value": "2147483647"
        },
        {
            "name": "taskMaxTime",
            "value": "9223372036854775807"
        },
        {
            "name": "priority",
            "value": "5"
        },
        {
            "name": "killCancelledTasks",
```

```
            "value": "true"
        },
        {
            "name": "checkpoint",
            "value": "false"
        },
        {
            "name": "unloadNativeLibrary",
            "value": "true"
        },
        {
            "name": "autoCancel",
            "value": "1"
        },
        {
            "name": "autoPackNum",
            "value": "0"
        },
        {
            "name": "collectionType",
            "value": "1"
        }
```

```
        ],
        "finished": true,
        "resultProperties": null,
        "driverUsername": "internal",
        "driverHostname": "10.128.88.108",
        "invocationInfo": [],
        "serviceId": 1185560448718947021,
        "completed": true,
        "submitTime": 1578979437654
    },
    {
        "finishTime": 1578980058243,
        "pendingCount": 0,
        "runningCount": 0,
        "completedCount": 1,
        "totalCount": 1,
        "cancelled": false,
        "serviceName": "Linpack Test",
        "description": [
            {
                "name": "appName",
                "value": "DataSynapse Job/Service Test"
```

```
        },
        {
            "name": "serviceTypeName",
            "value": "LinpackServiceTest"
        },
        {
            "name": "serviceName",
            "value": "Linpack Test"
        },
        {
            "name": "class",
            "value": "java:examples.linpack.LinpackService"
        },
        {
            "name": "serviceType",
            "value": "Service"
        },
        {
            "name": "slaGroupName",
            "value": ""
        },
        {
```

```
            "name": "providerType",
            "value": "JavaProvider"
        },
        {
            "name": "username",
            "value": "internal"
        }
    ],
    "status": "Finished",
    "options": [
        {
            "name": "autoPackMode",
            "value": "0"
        },
        {
            "name": "compressData",
            "value": "true"
        },
        {
            "name": "resultsPerMessage",
            "value": "100"
        },
```

```
        {
            "name": "resubmitOnDDTFailure",
            "value": "true"
        },
        {
            "name": "statusExpires",
            "value": "true"
        },
        {
            "name": "tasksPerMessage",
            "value": "20"
        },
        {
            "name": "deleteInvocationData",
            "value": "2"
        },
        {
            "name": "maxEngines",
            "value": "2147483647"
        },
        {
            "name": "taskMaxTime",
```

```
            "value": "9223372036854775807"
        },
        {
            "name": "priority",
            "value": "5"
        },
        {
            "name": "killCancelledTasks",
            "value": "true"
        },
        {
            "name": "checkpoint",
            "value": "false"
        },
        {
            "name": "unloadNativeLibrary",
            "value": "true"
        },
        {
            "name": "autoCancel",
            "value": "1"
        },
```

```
            {
                "name": "autoPackNum",
                "value": "0"
            },
            {
                "name": "collectionType",
                "value": "1"
            }
        ],
        "finished": true,
        "resultProperties": null,
        "driverUsername": "internal",
        "driverHostname": "10.128.88.108",
        "invocationInfo": [],
        "serviceId": 4863259086811987322,
        "completed": true,
        "submitTime": 1578980045220
    }
]"
```


service-invocation-count

Retrieves the total number of invocations of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/service-invocation-count
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
serviceId	Long	Service ID for which Service invocation count needs to be retrieved	3801168623022738886

Example Response

```
10
```

task-expiration-event-count

Retrieves the total number of task expiration events of the specified Service.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/task-expiration-event-count
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
serviceId	Long	Service ID for which total number of task expiration events need to be retrieved	3801168623022738886

Example Response

`0`

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/available
```

Example Response

`True` or `False`

list-resources

Lists resources on the Server.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/list-resources
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`staged`	Boolean	Staged flag. Can be true or false.	`False`

Example Response

```
[

    "calculator-1.0.0.1.tar.gz",

    "cppbridge-linux-gcc34-7.1.zip",

    "cppbridge-linux64-gcc34-7.1.zip",
```

```
    "cppbridge-linux64-gcc49-7.1.zip",
    "cppbridge-linux64-gcc83-7.1.zip",
    "cppbridge-win32-vc12-7.1.zip",
    "cppbridge-win32-vc14-7.1.zip",
    "cppbridge-win64-vc12-7.1.zip",
    "cppbridge-win64-vc14-7.1.zip",
    "cppbridge-win64-vc15-7.1.zip",
    "cppbridge-win64-vc16-7.1.zip",
    "jre-linux-1.8.0.311.tar.gz",
    "jre-linux64-1.8.0.311.tar.gz",
    "jre-win32-1.8.0.311.zip",
    "jre-win64-1.8.0.311.zip",
    "net5bridge-linux64-gcc49-7.1.zip",
    "net5bridge-win64-7.1.zip",
    "netbridge-7.1.zip",
    "netbridge-win64-7.1.zip",
    "pybridge-linux64-gcc34-7.1.tar.gz",
    "pybridge-win64-vc14-7.1.tar.gz",
    "python-linux64-3.7.0.zip",
    "python-win64-3.7.0.zip",
    "rbridge-linux64-7.1.tar.gz",
    "rbridge-win64-7.1.tar.gz",
```

```
    "SpeedLink-7.1.0.179676.tar.gz"

]
```

register-service

Registers the binding in the Service Registry; this must be done on the Primary Director.

Example Request

```
POST http://example.com:8080/livecluster/rest/service/register-service
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Service binding date that needs to be registered	{ "parameters": [{ "name": "classNa "value": "exampl JavaCalculator" }], "options": [{ "name": "gridLib "value": "calcul }

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE


```
    ],
    "description": [
        {
            "name": "service
            "value": "JavaCa
        },
        {
            "name": "service
            "value": "Java C
        },
        {
            "name": "provide
            "value": "JavaPr
        }
    ],
    "name": "JavaCalculatorE
    "uidescription": "Cross-
    "type": "java",
    "containerBinding": [
        {
            "name": "targetP
```

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
			"value": "exampl { "name": "xmlSeri "value": "false" { "name": "service "value": "*" }, { "name": "setStat "value": "setMem { "name": "appendS "value": "addToM] }"

Example Response

```
204 no content
```

Result: Service type is created.

all-finished-services

Removes the information of all finished Services from the Broker.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/service/all-finished-services
```

Example Response

`204 no content`

Result: All finished services are removed.

finished-service

Removes the information of finished Services from the Broker.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/service/finished-service
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Service ID for which finished Service needs to be removed	`3801168623022738886`

Example Response

`204 no content`

Result: Service with the specified ID is deleted.

resource-exists

Checks whether a file exists in the designated area.

Example Request

```
GET http://example.com:8080/livecluster/rest/service/resource-exists
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the resource that needs to be checked	`calculator-1.0.0.1.tar.gz`
`staged`	Boolean	Staged flag. True or false.	`false`

Example Response

`True` or `False`.

expires

Sets whether Service information is removed during Service cleanup.

Example Request

```
PUT http://example.com:8080/livecluster/rest/service/expires
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Job ID for which the expiration flag needs to be updated	`3001310211432770499`

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`expires`	Boolean	Expired flag. True or false	`true`

Example Response

`204 no content`

Result: Service information is removed during Service cleanup when set to true.

priority

Sets the priority of the specified Service.

Example Request

```
PUT http://example.com:8080/livecluster/rest/service/priority
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`jobId`	Long	Service ID whose priority needs to be set	`3001310211432770499`
`priority`	Int	Priority for the Service.	`2`

Example Response

`204 no content`

Result: Service priority is set to the specified value.

unregister-service

Unregisters the binding in the Service Registry; this must be done on the Primary Director.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/service/unregister-service
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the Service that needs to be unregistered	`JavaCalculatorExample`

Example Response

`204 no content`

Result: Service is removed from Service Types.

UserAdmin

UserAdmin APIs provide administrative access to the Users and Roles on a Manager. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
user	POST	Creates a new user with an initial password.
role	DELETE	Deletes the named role.
user	DELETE	Deletes the user with the given username.
all-roles	GET	Retrieves all roles.
all-users	GET	Retrieves all users.

METHOD	METHOD TYPE	DESCRIPTION
role	GET	Retrieves the named role.
user	GET	Retrieves the user with the given username.
available	GET	Retrieves whether the methods are available.
role	POST	Adds or updates a role.
user	PUT	Updates the user.

user

Creates a new user with an initial password.

Example Request

```
POST http://example.com:8080/livecluster/rest/user/user
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	User data that needs to be created	{ "username": "admin1", "firstName": "", "lastName": "", "email": "", "roles": "Configure" }

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`password`	String	Password for the user.	`admin1`

Example Response

`204 no content`

Result: User is created.

role

Deletes the named role.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/user/role
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the role you want to delete	`test`

Example Response

`204 no content`

Result: The specified role is deleted.

user

Deletes the user with the given username.

Example Request

```
DELETE http://example.com:8080/livecluster/rest/user/user
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
name	String	Name of the user to be deleted.	admin1

Example Response

`204 no content`

Result: The specified user is deleted.

all-roles

Retrieves all roles.

Example Request

```
GET http://example.com:8080/livecluster/rest/user/all-roles
```

Example Response

```
"[
    {
        "name": "Configure",
        "description": "The Configure role, all permissions.",
        "managerAccess": "*",
        "ldapGroup": "",
```

```
        "maxPriority": -1,

        "permissions": "Batch Manage,Batch Admin View,Batch Definition
View,Batch Registry View,Batch Definition Edit,Batch Schedule View,Broker
Admin View,Broker Admin Manage,Broker Configuration View,Broker
Configuration Manage,Broker Monitor,Broker Reports,Broker Routing
Manage,Broker Routing View,Credential Repository View,Credential
Repository Edit,Current Log,Diagnostics,Service Condition View,Service
Condition Edit,ExtraCondition Modify,Documentation,Driver View,Driver
Admin,Driver Download,Driver Events,Engine View,Engine Manage,Engine
Configuration View,Engine Configuration Edit,Engine Daemon View,Engine
Daemon Manage,Engine Events,Engine Install,Engine Log Files,Engine
Properties View,Engine Properties Edit,Engine Properties List View,Engine
Properties List Edit,Engine Daemon Reports,Event Subscription View,Event
Subscription Edit,Resource Deployment View,Resource Deployment
Maker,Resource Deployment Checker,GridCache View,GridCache
Manage,GridCache Schema View,GridCache Schema Edit,Grid Monitor,Server
Hook View,Server Hook Manage,Server Hook Edit,Import Export,Service
Session View,Service Session View Group,Service Session Manage,Service
Session Manage Group,Service Reports,License View,License Upload,Manager
Reconfigure,Manager Configuration View,Manager Configuration Edit,Service
Group View,Service Group Manage,Service Type List,Service Type
Manage,Service Type Edit,Service Type View,Service Runner View,Service
Runner Manage,SNMP View,SNMP Edit,Task Reports,Test Job,User Manage,User
View,User Events,Role View,Role Edit,Auth View,Auth Edit,Service Access to
All Users,Dashboard View,Dashboard Grid View,Dashboard Services
View,Dashboard Brokers View,Dashboard Drivers View,Dashboard Daemons
View,Dashboard Engines View,Web Services List View,Specify Additional
RunAs User,Execute Services,Quarantine Engine,Service Diagnostics"

    },

    {

        "name": "Manage",

        "description": "Management of all services and other components",

        "managerAccess": "*",

        "ldapGroup": "",

        "maxPriority": -1,
```

```
        "permissions": "Batch Manage,Batch Admin View,Batch Schedule
View,Broker Admin View,Broker Admin Manage,Broker Configuration
View,Broker Configuration Manage,Broker Monitor,Broker Reports,Broker
Routing Manage,Broker Routing View,Current Log,Diagnostics,Service
Condition View,Service Condition Edit,ExtraCondition
Modify,Documentation,Driver View,Driver Admin,Driver Download,Driver
Events,Engine View,Engine Manage,Engine Configuration View,Engine Daemon
View,Engine Daemon Manage,Engine Events,Engine Install,Engine Log
Files,Engine Properties View,Engine Properties Edit,Engine Properties List
View,Engine Properties List Edit,Engine Daemon Reports,Event Subscription
View,Resource Deployment View,Resource Deployment Maker,Resource
Deployment Checker,GridCache View,GridCache Manage,GridCache Schema
View,GridCache Schema Edit,Grid Monitor,Server Hook View,Service Session
View,Service Session View Group,Service Session Manage,Service Session
Manage Group,Service Reports,License View,Service Group View,Service Group
Manage,Service Type List,Service Type Manage,Service Type View,SNMP
View,Task Reports,Test Job,User View,User Events,Role View,Auth
View,Dashboard View,Dashboard Grid View,Dashboard Services View,Dashboard
Brokers View,Dashboard Drivers View,Dashboard Daemons View,Dashboard
Engines View,Web Services List View,Execute Services,Quarantine
Engine,Service Diagnostics"

    },

    {

        "name": "Service",

        "description": "View plus management of own services",

        "managerAccess": "*",

        "ldapGroup": "",

        "maxPriority": -1,

        "permissions": "Broker Admin View,Broker Configuration View,Broker
Monitor,Broker Routing View,Diagnostics,Service Condition
View,Documentation,Driver View,Driver Download,Engine View,Engine
Configuration View,Engine Daemon View,Engine Install,Engine Log
Files,Engine Properties View,Engine Properties List View,Resource
Deployment View,GridCache View,GridCache Schema View,Grid Monitor,Service
Session View,Service Session View Group,Service Session Manage,Service
```

```
Session Manage Group,License View,Service Group View,Service Type
List,Service Type View,Test Job,User View,Role View,Auth View,Dashboard
View,Dashboard Grid View,Dashboard Services View,Dashboard Brokers
View,Dashboard Drivers View,Dashboard Daemons View,Dashboard Engines
View,Web Services List View,Execute Services,Quarantine Engine,Service
Diagnostics"

    },

    {

        "name": "View",

        "description": "Default View Role",

        "managerAccess": "*",

        "ldapGroup": "",

        "maxPriority": -1,

        "permissions": "Broker Admin View,Broker Configuration View,Broker
Monitor,Broker Routing View,Service Condition View,Documentation,Driver
View,Engine View,Engine Configuration View,Engine Daemon View,Engine
Install,GridCache View,Grid Monitor,Service Session View,Service Session
View Group,License View,Service Group View,Service Type List,User
View,Role View,Auth View,Dashboard View,Dashboard Grid View,Dashboard
Services View,Dashboard Brokers View,Dashboard Drivers View,Dashboard
Daemons View,Dashboard Engines View"

    }

]"
```

all-users

Retrieves all users.

Example Request

```
GET http://example.com:8080/livecluster/rest/users/all-users
```

Example Response

```
[

    {

        "username": "admin",

        "firstName": "",

        "lastName": "",

        "email": "",

        "roles": "Configure"

    }

]
```

role

Retrieves the named role.

Example Request

```
GET http://example.com:8080/livecluster/rest/user/role
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the role whose information needs to be retrieved	`Service`

Example Response


```
{

    "name": "Service",

    "description": "View plus management of own services",

    "managerAccess": "*",

    "ldapGroup": "",

    "maxPriority": -1,

    "permissions": "Broker Admin View,Broker Configuration View,Broker
Monitor,Broker Routing View,Diagnostics,Service Condition
View,Documentation,Driver View,Driver Download,Engine View,Engine
Configuration View,Engine Daemon View,Engine Install,Engine Log
Files,Engine Properties View,Engine Properties List View,Resource
Deployment View,GridCache View,GridCache Schema View,Grid Monitor,Service
Session View,Service Session View Group,Service Session Manage,Service
Session Manage Group,License View,Service Group View,Service Type
List,Service Type View,Test Job,User View,Role View,Auth View,Dashboard
View,Dashboard Grid View,Dashboard Services View,Dashboard Brokers
View,Dashboard Drivers View,Dashboard Daemons View,Dashboard Engines
View,Web Services List View,Execute Services,Quarantine Engine,Service
Diagnostics"

}
```


user

Retrieves the user with the given username.

Example Request

```
GET http://example.com:8080/livecluster/rest/user/user
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
`name`	String	Name of the user whose information you want to retrieve	`admin`

Example Response

```
{

    "username": "admin",

    "firstName": "",

    "lastName": "",

    "email": "",

    "roles": "Configure"

}
```

available

Retrieves whether the methods are available.

Example Request

```
GET http://example.com:8080/livecluster/rest/user/available
```

Example response

`True` or `False`

role

Adds or updates a role.

Example Request

```
POST http://example.com:8080/livecluster/rest/user/role
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Role data that needs to be added or updated	{ "name": "Service1234", "description": "View plu "managerAccess": "*", "ldapGroup": "", "maxPriority": -1, "permissions": "Broker A }

Example Response

New role is created.

user

Updates the user.

Example Request

```
PUT http://example.com:8080/livecluster/rest/user/user
```

Example Input

Parameters

PARAMETER NAME	DATA TYPE	DESCRIPTION	SAMPLE VALUE
—	JSON	Role data that needs to be updated	{ "username": "admin", "firstName": "", "lastName": "", "email": "xyz@like.com", "roles": "Configure" }

Example Response

User data is updated successfully.

Version

Version APIs provide release version information. They are listed in the following table:

METHOD	METHOD TYPE	DESCRIPTION
version-release-name	GET	Retrieves release version.
build-version	GET	Retrieves build version.

version-release-name

Retrieves release version.

Example Request

```
GET http://example.com:8080/livecluster/rest/version/version-release-name
```

Example Response

```
7.1.1
```

build-version

Retrieves build version.

Example Request

```
GET http://example.com:8080/livecluster/rest/version/build-version
```

Example Response

`7.1.1.xxxxxx`

Upgrade Guide

- Upgrading GridServer
  - Requirements Changes
  - GridServer 7.1.1 Upgrade Checklist
    - Prior to Upgrade
    - After Stopping the Manager
    - During Installation
    - After Installation
  - Mixed-Version Deployments
  - Rolling Upgrades
    - Prerequisite Actions
    - Upgrading a Grid
  - Installing Hotfixes

Upgrading GridServer

This section explains how to upgrade GridServer components. It contains a GridServer Upgrade Checklist, which explains what to do before, during, and after your new GridServer installation to ensure carrying over old values to the new installation, when possible.

Requirements Changes

See the system requirements in the TIBCO GridServer® readme file to ensure that your software and hardware meet the minimum system requirements. Ensure to review the full list of requirements.

GridServer 7.1.1 Upgrade Checklist

You can migrate settings from your previous version of GridServer to your new version of a GridServer installation, but it’s important to not install on top of your previous installation. Instead, shut down your old Manager, and rename the old installation directory. You can then install the new version of GridServer in the same location. Later in the install, you supply a path to the renamed

old installation so settings can be copied. An `upgrade.log` file is generated that lists all changed configuration values, as well as settings that could not be migrated to the new installation.

It is recommended that before you start the upgrade, back up the `server.xml` file from the `DS_DATA/conf` directory to a separate directory to preserve your custom configurations. After the upgrade, replace the `server.xml` file in `DS_DATA/conf` with the updated file from the release. Then, reapply any required customizations.

The only direct upgrade supported on GridServer version 7.1.1 is from GridServer versions 7.1.0. To upgrade from any other earlier version to GridServer 7.1.1, you must first upgrade to GridServer version 7.1.0.

Note:

- .

  Authenticating all Grid Components is an additional security option provided in GridServer. For more information, see Authentication of Grid Components.
- Migrate any deployed web components, Broker or Manager hooks, or bundled libraries that reference the `javax.*` servlet API to the `jakarta.*` namespace.

Prior to Upgrade

- If you have any UNIX Engines that share a common installation directory, you must migrate those installations to a local directory. A shared install point is no longer supported.
- To enable programmatic version control, ensure that `gridLibraryStrictVersioning` is set to `false`.
- If you have a Linux64 Engine, for upgrade from all versions earlier than 7.0.0_hotfix08, you must perform the following steps:

1. Stop the server.
2. As a best practice, take a backup of the existing `engine.sh` file.
3. Replace the `engine.sh` file from the `engineScriptUpdate` folder to `datasynapse/manager/webapps/livecluster/engineUpdate/linux64/ and datasynapse/manager-data/engineUpdate/linux64/` folder.
4. Start the server.

5. Let the Engines log in and then restart the Engine Daemon.
6. Stop the server.
7. Upgrade the server.

> **Note:**In case of Grid Architecture, perform the steps from step a to step g on the Primary Director.

- Check the list of changes in the release notes for any other issues that might affect you.

After Stopping the Manager

- When running GridServer as a Windows service, remove the service prior to upgrading.

  The following steps are necessary only if you set up the system by using an external database:

- Copy the corresponding database properties file from old `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db/` directory to the current installation. This is used when you run the database upgrade script.
- Upgrade your external reporting database schema using the `upgradedb.sh` or `upgradedb.bat` script located in `DS_MANAGER/webapps/livecluster/WEB-INF/etc/db`.

During Installation

- For **Installation Type**, choose **Custom, Manager Upgrade**.
- For **Previous Data Directory**, enter the pathname of your previous base directory.

After Installation

If Using System Classloader for Engines

If you are using the system classloader in your Engine configuration and deployed the `DSEngine.jar` file to Engines, deploy the new `DSEngine.jar` to Engines after upgrade. This is now deployed in `DS_MANAGER/webapps/livecluster/engineUpdate/shared.` If `DSEngine.jar` is deployed in the default resources directory, it has already been removed. If it is deployed to any Grid Libraries, you must remove it from them.

External Database JDBC Drivers

If you are using an external database, copy the required JDBC driver JAR archive from `DS_MANAGER/webapps/livecluster/WEB-INF/lib` in the old installation to `DS_MANAGER/webapps/livecluster/WEB-INF/lib/` in the new installation.

Apply License Using Script

For an existing installation, run the `apply-license` script after the installation completes and before you start the Primary Director. The `apply-license` script is at the following location:

Windows: `DS_MANAGER\apply-license\apply-license.bat <license.ser>`

Linux: `DS_MANAGER/apply-license/apply-license.sh <license.ser>`

> **Note:**
>
> - Ensure that the Primary Director is not running when you run this script. If Tomcat is already started, it holds the database file lock, which causes the script to fail.
> - You can obtain the license file from DataSynapse Support.

LDAP Upgrade

Starting with GridServer 6.0, Driver profiles are no longer supported. The `allowedBrokers` property has been replaced by the value of **Manager List**in the security role assigned to the Driver user. Also, users are now allowed to run Services based on the "Execute Services" permission rather than by virtue of having a profile.

During upgrade, a new role is created for each Driver profile. This role has execute permission, and maps the "Allowed Brokers" field to "Manager List". That role is added to each user who has the corresponding profile.

You are not required to maintain these converted profiles; you might choose to reorganize and remap these roles as you see fit.

If you use LDAP authentication and you prefer to assign roles via LDAP groups, you must add those groups to your users in LDAP, and remove the locally assigned roles.

Users and roles are logged during the upgrade to the file `DS_DATA/logs/newRoles.txt`, so that you have a list of groups to add the user's LDAP entries. It is a flat file with the following format:

```
[user]\t[role1,role2,...]
```

For example:

```
joe driver_developers,driver_boston

tom driver_testing,driver_london
```

Engine Upgrade

- For UNIX Engines, `bin/invokeRA` cannot be automatically upgraded. After upgrading, you must manually copy the new `bin/invokeRA` to the upgraded Engine, and follow the instructions in "Using Run-As" in the TIBCO GridServer® Administration.

Driver Upgrade

- Download the new version of the GridServer SDK in the GridServer Administration Tool.
- Make sure that all Drivers are upgraded. This simply involves replacing the libraries or executables with the new version.
- If you have existing CPPDriver applications, relink them with the new libraries included in the SDK.
- On the Driver side, link your code with the `dsUtil` library. Link Engine-side Service code with the `dsUtil` library if it uses any exception classes.

Note:

After the server is upgraded, you can secure the `Internal.script` and passwords stored in the `installation.properties` file at `DS_DATA\conf`. For more information, see Securing Passwords and Internal.script in the TIBCO GridServer® Manager Installation Guide.

Mixed-Version Deployments

Mixed-version deployments are grids that consist of GridServer Manager versions 7.1.0. This is an allowed configuration, as a convenience for large Grids that cannot upgrade all Brokers at once. You must address the following issues when running mixed-version deployments:

- The Directors must be upgraded first to the latest version of GridServer Manager 7.1.0.
- All Brokers must be on the latest service pack and hotfix of the minor version.
- You must disable client version checking for Engines, Drivers, and Brokers at **Admin > System Admin > Manager Configuration > Admin**.

- Hard-partitioning of Engines is required. That is, Engines must not be allowed on the GridServer Manager version 7.1.0 Broker. This is done with the balancer configuration.
- You must deploy all versions of the C++ Bridge Grid Libraries via the Primary Director. If you upgraded the Director, the older bridges might already be there, but if the Director is a new installation you must copy them from your old Managers. The Engines automatically choose the proper library based on the Broker version. Because both versions of the C++ Bridges are used, Grid Libraries must also be used on an older Broker if the code is native, rather than using the standard resource deployment.

Mixed-Version Driver Compatibility

GridServer Manager 7.1.1 has backward compatibility with GridServer Manager 7.1.0.x Drivers.

Rolling Upgrades

Use the following procedures to install an update to a large grid (a grid with a large number of Engines and several Brokers) in situations where you cannot permit grid-wide Engine restarts and interruptions in Service availability. These instructions enable Services already deployed on components using a previous version of GridServer Manager to continue running as other components on the grid are updated.

Prerequisite Actions

For all Managers:

1. Bring all Managers up to the required hotfix level.
2. Confirm that all GridServer Managers are at that level. On the Primary Director, you can get the versions of all Brokers on the **Grid Components > Brokers > Broker Admin**page. To find the version of the Directors, click the **About**link at the top of the Administration Tool.
3. In each Engine Configuration, confirm that **Log off Engines on Daemon Logoff**is set to false (the default setting). This prevents unnecessary Engine restarts when the Director is restarted.
4. On each Broker, go to **Admin > System Admin > Manager Configuration > Admin**and set **Daemon Upgrade From Broker**to false; then go to **Admin > System Admin > Manager Configuration > Resource Deployment > Broker Settings**and set **Synchronize Resources From Director**to false. This prevents all Daemons from restarting to upgrade when the Primary Director is upgraded.
5. On each Director, go to **Admin > System Admin > Manager Configuration > Admin**. Confirm that **Daemon Upgrade From Director**is false. (This is the default and would rarely be true.)

6. To assist in the event of any issues, set the logging levels on all Managers and Engines to Fine.

Upgrading a Grid

Note that each Manager upgrade procedure does not make any modifications to the previous installation. If required, you can use this directory to roll back the upgrade.

Primary Director Upgrade

1. Shut down the Primary Director. The Engine Daemons move to the Secondary Director and the Engine Instances (which might be running tasks) continue without interruption.
2. Upgrade the Primary Director.
3. Start the Primary Director.
4. Verify the the following Director and Broker mismatch controls are True: **Admin > System Admin > Manager Configuration > Admin > Version Management >Allow Director Version Mismatch**and **Admin > System Admin > Manager Configuration > Admin > Version Management >Allow Broker Version Mismatch**.
5. Check that all the Brokers are logged in to it at **Grid Components > Brokers > Broker Admin**. The Engine Daemons move to the Primary Director. The Engine Instances do not restart.

Secondary Director Upgrade

Follow the above procedure for the Primary Director. Shut down the Secondary Director and upgrade.

Broker Upgrades

Failover Brokers must be upgraded after regular Brokers.

For each Broker:

1. Shut down the Broker. Within a few minutes Engines and Drivers migrate to a Failover Broker.
2. Upgrade the Broker.
3. Start the Broker. Verify that it logs in to the Directors.
4. Verify that **Admin > System Admin > Manager Configuration > Admin > Version Management > Allow Driver Version Mismatch**is True.
5. Within a few minutes Engines and Drivers migrate back.

Upgrade the Engine Daemons

At this point, the Grid has been updated, but Engine Daemons are still at the previous version. Daemon upgrades result in Engine restarts, so do the following on each Broker when the time is appropriate:

- On each Broker, go to **Admin > System Admin > Manager Configuration > Admin**. Set **Daemon Upgrade From Broker**to true.

> **Note:**The above step needs to be done as soon as possible.

Installing Hotfixes

To install or uninstall a hotfix or service pack, follow the instructions in the readme file distributed with the hotfix.

Security Guide

- Security Guidelines

Security Guidelines

Introduction

This document is meant as a guide to implement security features for GridServer Manager. The goal is to provide you with information on what can be done and where you can go to find the information necessary to implement the solutions.

As GridServer uses third-party products, what can be done with those products is likely to change over time. Therefore, it is necessary for you to continue to stay educated on the latest features of third-party products.

This document is specific to GridServer 7.1.1. and assumes that the reader has a working knowledge of GridServer components. If not, see the TIBCO GridServer® Introducing TIBCO GridServer® documentation.

Product Connectivity

This section describes the connectivity between GridServer components. For a general explanation of network topography, see the Configure Your Network topic in the TIBCO GridServer® Installation.

Communication

- Director-Broker Communication

  The communication between the Director and Broker uses TCP on port 5635 by default. See the Configure Your Network topic in the TIBCO GridServer® Installation.

- Manager-Driver Communication

  The ports used by the Manager for incoming communication from the Driver are covered in the Configure Server Settings topic in the TIBCO GridServer® Installation.

- Manager-Engine Communication

The ports used by the Manager for incoming communication from the Engine are the same as the ports used for the Driver. See Director-Broker Communication.

- Driver-Engine Communication

  The Driver and Engine communicate directly when either of them has Direct Data Transfer (DDT) enabled. If the Driver has DDT enabled, then the Engine attempts to download input data directly from the Driver's file server. If the Engine has DDT enabled, then the Driver attempts to download the output data directly from the Engine's file server.

  An overview of DDT can be found in the Optimizing the Grid topic in the TIBCO GridServer® Administration. More information is available in the Diagnosing Network Issues topic of the same guide.

- Driver DDT

  Disabling Driver DDT can be found in the Direct Data Transfer section in the Optimizing the Grid topic in the TIBCO GridServer® Administration.

  Driver Cleaner Configuration can be found in the Driver Installation topic in the TIBCO GridServer® Developer's Guide.

- Engine DDT

  Disabling Engine DDT is found in the Direct Data Transfer section in the Optimizing the Grid topic in the TIBCO GridServer® Administration.

  For information about specifying the Engine's `ds.DataDir` value, see the Invocation Variables topic in the TIBCO GridServer® Developer's Guide

- DDT KB Articles

  KB Article 000008671: Using static ports in Engine and Driver DDT settings.

SSL

Enabling SSL is covered in the Configuring SSL topic in the TIBCO GridServer® Installation.

SSL KB Articles

- KB Article 000043090: How to configure GridServer to use JSSE SSL implementation instead of OpenSSL
- KB Article 000036224: Steps for creating SSL certificates that can be used on GridServer Manager

- KB Article 000031710: Steps to enable extra logging to debug SSL connection errors

Authentication and Authorization

- Run-As

  There are often cases where Services require specific user permissions to access needed resources. By creating the Engine process as a given user, all Service invocations that are run by the Engine can operate with these permissions.

  - Running windows engine as Run-AsConfiguring the Windows engine as Run-As is covered in the Windows Engine Installation topic in the TIBCO GridServer® Installation.
  - Running UNIX engine as Run-As Configuring the UNIX engine as Run-As is covered in the UNIX Engine Installation topic in the TIBCO GridServer® Installation.
  - Running service as Run-AsConfiguring Service as Run-As is covered in the Service Run-As topic in the TIBCO GridServer® Administration.

- NTLM/Kerberos

  Pure Kerberos authentication allows windows managers to enable authentication without requiring a password.For more information about Pure Kerberos authentication, see Configuring Pure Kerberos Authentication topic in the TIBCO GridServer® Administration Guide.

- LDAP Authentication and Authorization

  LDAP can be used for:

  - Authentication, where a user name is authenticated against a directory entry;
  - Authorization, where groups assigned to that user map to GridServer roles
  - Both authentication and authorization.

- Role-Based Access Control

  GridServer supports a customizable system of role-based access control to provide account security and enable different users to access different areas of the interface. This is covered in the The GridServer Administration Tool topic in the TIBCO GridServer® Administration.

- Engine Daemon Authentication

  Engine Daemon authentication can be enabled during the Engine installation. This is covered in the Enabling Engine Daemon Authentication topic in the TIBCO GridServer® Installation.

- Authentication of Grid Components

  You can choose to enable Authentication between all Grid components. This is covered in the Authentication of Grid Components topic in the TIBCO GridServer® Installation.

- Reporting Database Authentication (on-premise AD)

  When you use Microsoft SQL Server as the reporting database, the GridServer Manager can authenticate its database connection using on-premise Active Directory credentials over NTLM. This method provides an alternative to SQL login and password. This database connection authentication is distinct from the user and grid-component authentication to the Manager.

  For information about configuration steps, see "Configuring Active Directory (AD) Authentication for the SQL Server Reporting Database (On-Premise)" section in the TIBCO GridServer® Installation guide.

- Reporting Database Credentials through HashiCorp Vault

  GridServer allows you to retrieve the Reporting Database username and password from a HashiCorp Vault server at connection time, eliminating the need to store them directly within the Manager configuration. For more information, see "Configuring HashiCorp Vault for Reporting Database Credentials" section in TIBCO GridServer® Installation guide.

Web UI Features

- Single Sign-On

  After logging in once to any Manager, you can use the Grid Single Sign-On (SSO) feature to log in to all Managers on a Grid.Configuring web Single Sign-On is covered in the Grid SSO Configuration topic in the TIBCO GridServer® Administration.

- Quarantine Broker

  When the Quarantine Broker is specified, the Director allows individual Engine instances to log in to either production Brokers or the Quarantine Broker. It is based on the value of the quarantine status of the managing Engine Daemon. Fore more information about configuring web Single Sign-On, refer to Quarantine Brokers topic in the TIBCO GridServer® Administration.

Updating Third-Party Components

Before updating any third-party components, TIBCO recommends that you contact TIBCO Support to determine if a specific version of a third-party component has been tested with GridServer. While service pack updates generally work, some major or minor version updates can often result in unexpected behavior or exceptions.

- Configuring Security Options
  - Apache Tomcat

    You must be familiar with the many considerations and options related to Tomcat security. Tomcat has its own Security Guide and you are encouraged to review it.
  - XSS Filter

    The XSS Filter on the Manager can be enabled or disabled by modifying the associated property in the GridServer Administration Tool at **Admin > Manager Configuration > Security > Miscellaneous**.To edit the whitelist, open the `xss.xml` file located at `DS_INSTALL/manager/webapps/livecluster/WEB-INF/config/`
  - CSRF Filter

    The CSRF Filter on the Manager can be enabled or disabled by modifying the associated property in the GridServer Administration Tool at **Admin > Manager Configuration > Security > Miscellaneous**.

    To edit the whitelist, open the `xss.xml` file located at `DS_INSTALL/manager/webapps/livecluster/WEB-INF/config/`
- Disabling directory traversal on the File Server port

  To disable directory traversal for the Engine file server, update the `Engine.xml` file with the following code:

```
<contexts c="PropertiesBean" n="Context">
 <property n="object" v="DSDataDir" k="data"/>
 <property n="object" v="DSWorkDir" k="work"/>
 <property n="object" v="$DSProfileDir$/logs" k="logs"/>
```

```
</contexts>
```

- Setting up User Authentication - This is covered in the following guides:
  - TIBCO GridServer® Administration - Configuring GridServer for LDAP Authentication or Authorization topic.
  - TIBCO GridServer® Installation - Grid Components Authentication and Enabling Engine Daemon Authentication topics.

Enabling the Encryption of Task Payloads

To secure the data that is being passed between Drivers and Engines, it is possible to enable encryption for the task payload. This is done by editing the Service Type as follows:

1. In the GridServer Administration tool, select the **Services**menu.
2. Select **Service Types**.
3. Edit one of the Service Types options.
4. The feature is listed in the Options section for **Service Type**and it is called `encryptionEnabled`.

COM Integration Tutorial

- Accessing a Service from Visual Basic
  - Getting Started
    - Installing the GridServer SDK
    - Setting up the Service
    - Referencing COMDriver in the Visual Basic Editor
  - Client-side Code
    - Creating a Service Instance
    - Submitting Requests Synchronously
    - Submitting Requests Asynchronously
    - Collecting Results During Asynchronous Execution
    - Stateful Service Operations
    - Discriminators
    - Setting Driver Properties
    - Destroying Service Instances
- Excel Demo
  - Getting Started
    - Enabling Macros in Excel
    - Deploying Resources
    - Registering the Service Type
    - Executing the Demo
    - The Visual Basic Code

Accessing a Service from Visual Basic

This guide is your reference for developing COM applications that use TIBCO GridServer® installations. It is divided into several sections covering the fundamentals of how to use the GridServer COM API.

In this section, we demonstrate how to access a Service from a client application written in Visual Basic, using COMDriver. The example shows how a Visual Basic client can call COMDriver APIs to create and destroy a Service instance, run the Service synchronously or asynchronously, and update the state of the Service. Although the Service in this example is a Java Service, client applications using COMDriver can access Services written in any server-side compatible language, provided that Service methods use strings for arguments and return values.

Getting Started

Installing the GridServer SDK

Before you begin, you must download and install the GridServer SDK, which is available from your Manager in the GridServer Administration Tool. The SDK includes the COMDriver. For more information about installing the SDK, see the *TIBCO GridServer® Developer's Guide*.

Getting Started

Installing the GridServer SDK

Before you begin, you must download and install the GridServer SDK, which is available from your Manager in the GridServer Administration Tool. The SDK includes the COMDriver. For more information about installing the SDK, see the *TIBCO GridServer® Developer's Guide*.

Setting up the Service

The Service in this example simulates a simple calculator, with functions for the four operators, plus a memory function. The Java source code is available in the GridServer SDK, in `/GridServerSDK-win32/examples/service/calculator/service/java/`.

The `JavaCalculator` class has a floating-point field called `memory`, and the following methods:

- `add`, `subtract`, `multiply`, `divide` — each takes two strings representing floating-point numbers, and returns a string representing the result of the arithmetic operation.
- `addToMemory` — takes in one string argument representing a floating-point number, and stores the sum of the input and `memory` values back in `memory`.

- `setMemory` — takes in one string argument representing a floating-point number and sets the `memory` field equal to it.
- `getMemory` — takes in a string argument, and returns the value of `memory`. (The argument is unused, but is required by the Services API.)

Before the client can access the Service, we have to register a Service Type and deploy the Service resources. Make sure the Service Type is registered under the name `JavaCalculatorExample` in the GridServer Service Type Registry; this is how we reference the Service from the client. The Service is automatically registered when GridServer is installed.

If the Grid Library has not been deployed, build and deploy it from `GridServerSDK-win32/examples/service/calculator/` by running `build-and-deploy.bat`.

To find out more about writing, registering, and deploying a Service, see the *TIBCO GridServer® Service-Oriented Integration Tutorial*.

Referencing COMDriver in the Visual Basic Editor

The COMDriver library has to be referenced in the Visual Basic client application. With the client application opened in Microsoft Visual Basic Editor, go to the **References**dialog. Make sure that the reference to **DataSynapse Driver 1.0 Type Library**is checked. It must be referenced to `[COMDriver installation]\DSCOMDriver.exe`.

Client-side Code

Now that the Service has been registered and deployed, we can access it from the client through COMDriver. Let's start by creating an instance of the Service.

Creating a Service Instance

To create an instance of the Service `JavaCalculatorExample` using COMDriver, use the `createService` function in the `DSCOMDRIVERLib` package.

```
Dim csfactory As New DSCOMDRIVERLib.ServiceFactory

Dim service As DSCOMDRIVERLib.service

Set service = csfactory.createService("JavaCalculatorExample", "0",
Nothing, Nothing)
```

The first line creates an instance of `ServiceFactory`. This is used to create TIBCO GridServer® Service instances. `ServiceFactory` is defined in the `DSCOMDRIVERLib` library, which resolves in `DSCOMDriver.exe`.

The second line declares a variable called `service` that holds the Service instance. This variable must be visible to any other part of the code that wishes to interface with the Service. You can see that the rest of the code in this section references this variable.

The third line calls the `createService` method, which takes four arguments:

- `serviceName` — The name of the registered Service (String).
- `initData` — The state initializing data for the Service (String).
- `options` — Service options; see the `Options` class in the `Service` JavaDoc for available options.
- `description` — Service descriptions; see the `Description` class in the `Service` Javadoc for available description options.

The third and fourth arguments are of type `Properties`, which is a set of name-value pairs. Consult the COMDriver documentation for details.

The method returns the created Service interface. In our example, an instance of the `JavaCalculatorExample` Service is initialized with the value `0`, with no options and description.

If the supplied Service name is not registered on TIBCO GridServer®, the application fails with a runtime error: `Service: servicename.csdd not found`.

Submitting Requests Synchronously

Work requests can be submitted to a Service synchronously or asynchronously. Let us go through submitting a request synchronously first. Depending on the number of input arguments you have for the `Service` function, you can use the `execute` or `executeWithArray` function.

Both `execute` functions take three arguments:

1. `methodName` — the name of the method to invoke.
2. `data` — the input data.
3. `discriminator` — the discriminator used for this request; discriminators are discussed in a later section.

Both return the output data as a string. The input and output arguments are restricted to strings, for reasons of network serialization, and better interoperability between clients and Services of different languages.

The only difference between the two functions is that `executeWithArray` accepts an array of strings as the input data argument, whereas `execute` accepts a single string.

For example:

```
Dim numInMemory As String

numInMemory = service.execute("getMemory", "", Nothing)

Dim sum As String

Dim args(1) As String

args(0) = "25"

args(1) = "75"

sum = service.executeWithArray("add", args, Nothing)
```

If the Engine-side function that is being run takes one or no arguments, it is more convenient to use `execute`; otherwise, place your arguments in an array, and pass them into `executeWithArray`. If the function takes no arguments, like the `getMemory` function, use `execute` with an empty string.

The `execute` functions do not return until the Service completes the request and returns the result. To have the function return immediately, use the `submit` function to submit requests asynchronously, as described below.

Submitting Requests Asynchronously

To submit requests to the Service asynchronously, use the `submit` functions. They follow the same format as `execute`. One of the two functions is more suitable depending on the number of input arguments the Service function takes: `submit` for zero or one argument, and `submitWithArray` for more than one argument.

The `submit` functions return immediately, with the invocation ID (a Long integer) as the output argument. The ID increments sequentially and starts at 0; it can be used to identify results when they are later collected.

```
Dim ID1 As Long

ID1 = service.submit ("getMemory ", "", Nothing)

Dim ID2 As Long

Dim args(1) As String

args(0) = "25"

args(1) = "75"

ID2 = service.submitWithArray("add", args, Nothing)
```

The results are collected by calling the `collectNext` function.

Collecting Results During Asynchronous Execution

Since Visual Basic is not multithreaded, there are no callbacks for results collection. The `collectNext` function is used for collecting results during asynchronous calls. The function returns one of the available results from the Service; if none are available, it waits for the number of milliseconds supplied as an input argument before returning. The timeout prevents a client application from waiting indefinitely on a Service.

`collectNext` returns the result as a `ServiceData` type, which includes the following properties:

- `Id` — The Service invocation ID, which can be used to match the result with the corresponding submission.
- `IsError` — A Boolean that signifies whether the execution resulted in an error; if so, the data property contains the error message.
- `Data` — The result (Service function returns value or error).

For the same reason as `execution` and `submit` functions, all output arguments are pure strings.

To make collecting multiple results easier, the `InvocationCount` parameter of a Service is equal to the number of outstanding invocations.

The following code demonstrates how one can implement results collection. It repeatedly checks a Service for available results. If there are results, one result is collected; the code checks to see if the result is an error, and prints out the result data accordingly. If there are no results, it waits for 1 second, and then loops. The loop ends when all the results have been collected.

```
While service.InvocationCount > 0

    Dim sd As ServiceData

    Set sd = service.collectNext(1000)

    If Not sd Is Nothing Then

        If sd.IsError Then

            Debug.Print "collected " & sd.Id & " error: " & sd.Data

        Else

            Debug.Print "collected " & sd.Id & ": " & sd.Data

        End If

    End If

Wend

Debug.Print "done collecting"
```

Stateful Service Operations

To better understand how to manage state from the client side, let us take a quick look at how Services on TIBCO GridServer® manage it. Let's consider our `JavaCalculatorExample` Service.

The state maintained by the Service is a field called `memory.` There are two methods that are responsible for updating the state:

- `addToMemory`

- `setMemory`

  Although both methods update the state, TIBCO GridServer® has to handle them differently to ensure that the state is properly maintained. `addToMemory` refers to the previous value of `memory`, whereas `setMemory` does not. TIBCO GridServer® denotes the first type of method as an `appendStateMethod`, and the second type as `setStateMethod`.

  When a Service instance is created on an Engine, the Engine replays all state-changing methods from the last `setStateMethod`, or from the beginning if there have been no `setStateMethods`. Depending on the duration of the request, it might be worthwhile to follow `appendStateMethods` with `setStateMethods` periodically, to save memory on the Manager.

  These special state-changing methods have to be identified when the Service is registered in TIBCO GridServer®. Within the Service configuration of the `JavaCalculatorExample`, under the section **ContainerBinding**, these method names must be included in the fields titled **appendStateMethods**and **setStateMethods**.

  Once the methods have been registered on the Service, a client can access them only through the `updateState` functions. These functions behave similar to the `execute` functions. They take three input arguments:

- `methodName` — the name of the method to invoke.
- `data` — the input data.
- `append` — a Boolean signifying whether it is an `appendStateMethod` and updates the state accordingly.

These functions do not return output arguments.

In the same manner as `execute` and `submit`, one of two `updateState` methods are more suitable depending on the number of input arguments that the Service function takes. These functions are: `updateState` and `updateStateWithArray`.

The following Visual Basic code updates the `memory` of the `JavaCalculatorExample`:

```
service.updateState "setToMemory", "25", False

service.updateState "addToMemory", "75", True
```

The resultant value of `memory` is 100.

Discriminators

Discriminators allow Service requests to be performed only on Engines that meet the discriminator's requirements. To construct a discriminator, use the COM types `PropertyDiscriminator` and `PropertyComparator`. A `PropertyDiscriminator` is a collection of `PropertyComparators`, each of which expresses one requirement on Engines that must evaluate to true in order for the engine to qualify. See the COMDriver API reference for a full list of comparators and machine properties; it is located at `[COMDriver installation]/docs/COMDriver/DSCOMDriverRef.doc`.

The following Visual Basic code creates a new `PropertyDiscriminator`, which limits operation to machines with more than 200MB of free disk space, and the custom property `foo` equal to the value `bar`. It then submits a request to the `add` function with the `PropertyDiscriminator`.

```
Dim discrim As New PropertyDiscriminator

Dim pc As New PropertyComparator

pc.Init FREE_DISK_MB, "200", GREATER_THAN_EQ, False

discrim.Add pc

pc.Init "foo", "bar", EQUALS, False

discrim.Add pc

Dim args(1) As String

args(0) = "25"

args(1) = "75"

ID2 = service.submitWithArray("add", args, discrim)
```

The `Init` function of `PropertyComparator` initializes the comparator as comparison with a predefined Engine property, and takes four arguments:

- `name` — A string representing the name of the Engine property.
- `value` — A string representing the value of the Engine property.
- `op` — The comparison method, from the enumeration `DSComparatorType`.

- `nullCompare` — The result of the comparison when the value of the property is not defined on the Engine.

For more information about initializing `PropertyComparator`, see the COMDriver API reference (`[COMDriver install dir]\docs`). For more information about custom properties for Engines, refer to the “Using Conditions” topic of the *TIBCO GridServer® Developer’s Guide*.

Setting Driver Properties

COMDriver obtains its configuration from a file called `driver.properties`. It is possible to override the values in `driver.properties` programmatically, using the `DriverManager` class. This ability is most commonly used to set a user name and password for authentication to the Grid. Setting these values in the code allows different users to reference the same `driver.properties` file, and avoids having a cleartext password in the file. To set the user name and password in your code, place the following lines before any other use of COMDriver. This code assumes that `user` and `password` are variables that have been set to the desired user name and password.

```
Dim dm As New DSCOMDRIVERLib.DriverManager

dm.setProperty DSUSERNAME, user

dm.setProperty DSPASSWORD, password
```

Destroying Service Instances

When clients have completed submitting requests to a Service instance, it is responsible for destroying the instance to free resources on the server. To do so, simply call the destroy function on the Service. Any outstanding invocations are canceled.

Excel Demo

This section details an example of how Windows applications can interface with TIBCO GridServer® Services through COMDriver and Visual Basic. The client application is a Portfolio Valuation calculation in Microsoft Excel, and provides two ways of performing the calculation: locally in Visual Basic, or as a Java Service on TIBCO GridServer® using COMDriver.

The example is available from the COMDriver installation, and is located in `[DSCOMDriver installation]/examples/exceldemo`.

Getting Started

Before running the demo, you must enable Macros in Microsoft Excel, deploy the resources to TIBCO GridServer®, and register the Service Type.

Enabling Macros in Excel

By default, macros are not enabled in Excel spreadsheets. Enable them as follows:

1. Start Excel.
2. In Excel, choose **Tools > Macros > Security**.
3. Select **Medium Security**.

Deploying Resources

To deploy the resources used by the spreadsheet to your Engines, in the COMDriver installation, upload the Grid Library ZIP file in `examples/exceldemo/service/` directory in the GridServer Administration Tool on the **Grid Components > Services > Grid Libraries**page.

Registering the Service Type

After deploying the Excel demo, you must register the Service Type as follows:

1. Go to **Services > Services > Service Types**.
2. In the empty box at the bottom of the table, enter “exceldemo” and select **java**for the implementation. A new window appears with options for registering a Service.
3. In **className,**enter `exceldemo.TradeValuationService`.
4. In **Options**, set the following:
5. Set **autoPackNum**to 5.
6. Set **gridLibrary**to `exceldemo`.
7. Set **gridLibraryVersion**to `1.0.0.1`.
8. Click **Save**.

Executing the Demo

Open the Excel file `OptionValuationDemo.xls`, and when prompted, enable Macros. After several seconds, a message appears that says: `GridServer connection established`. Click **OK**. If this does not occur, make sure the Manager is running and visible from your machine, and that you supplied the Manager name and port when you installed COMDriver.

To run the demo, either select **Use Local Spreadsheet**to run it locally, or **Use DataSynapse Grid**, to enable Excel to access the Java Service on GridServer, and then click **Value Portfolio**.

Note that the calculation completes faster if you select the **Use DataSynapse Grid**option, because calculations in Visual Basic are much slower than in Java.

The Visual Basic Code

To see the Visual Basic code within the Excel file, from the **Tools**menu, select the **Macro**menu, then click **Visual Basic Editor**. This opens up a Visual Basic Editor; select **Sheet1**in the Project Explorer to see the source code.

The code must be familiar to you if you read through the last Visual Basic example. On loading the Excel file, the `InitializeLC()` function is performed. It instantiates `ServiceFactory`, and creates an instance of the Service `exceldemo`.

```
Public Service As DSCOMDRIVERLib.Service

Sub InitializeLC()

    Dim f As New DSCOMDRIVERLib.ServiceFactory

    On Error GoTo iError

    If (Service Is Nothing) Then

        Set Service = f.createService("exceldemo", "", Nothing, Nothing)

    End If

    MsgBox ("GridServer connection established.")

    GoTo iEnd

iError:      MsgBox ("GridServer Service not found: " & "(" &
Err.Description & ".)")

iEnd:

End Sub
```

The Visual Basic function that runs the calculation on GridServer is called `runLiveCluster`. It calls the Java function `valueTrade` of the Service asynchronously using the `Submit` function in COMDriver, sequentially from the first to the last row. The function takes one string argument: all the values on the row of the deal, concatenated into one string, delimited by spaces.

The code correctly assumes that Invocation IDs returned by the `submit` function are sequential. The invocation ID of the last call is written to the variable `result`. This value is important for indexing the results later on.

```
For rwIndex = begRow To endRow

Dim a As String

a = Cells(rwIndex, 2).Value & " " & Cells(rwIndex, 3).Value & " " _

& "0.05 " & Cells(rwIndex, 4).Value & " " & Cells(rwIndex, 5).Value & " "
_

            & Cells(rwIndex, 6).Value & " " & Cells(rwIndex, 7).Value & "
" & sims

Dim result

result = Service.submit("valueTrade", a, Nothing)

Next rwIndex
```

Since `submit` is an asynchronous call, the results are collected separately. The code calls the `collectNext` function repeatedly until the number of results collected is equal to the number of rows. Alternatively, it can use `Service.InvocationCount`, to determine if there are still pending results. See the previous example for an implementation of this.

The row number is calculated by offsetting the invocation IDs by the variable `result`:

```
'   now collect

Dim count As Integer

While count < endRow - begRow + 1

Dim data As ServiceData
```

```
Set data = Service.collectNext(100)

If Not (data Is Nothing) Then

Cells(data.ID + begRow - (result - (endRow - begRow)), 7).Value =
data.data

count = count + 1

End If

Call updateProgress(count * 100# / (endRow - begRow + 1))

Wend
```

PDriver Tutorial

- Using PDriver Tools
  - Product Overview
  - Before You Begin
  - Running PDriver
    - Running Synchronously
    - Running Asynchronously
    - Running Commands
    - Other PDriver Commands
  - Summary
- PDS Structure and Variables
  - PDS Structure
    - The Basic Script
    - The Options Block
    - The Schedule Block
    - The Discriminator Block
    - Prejob and Pretask Blocks
    - Posttask and postjob blocks
  - Variables and Parameters
    - The Basics
    - About Scope
  - Summary
- PDS Statements
  - Built-in statements
    - Command Execution

        - File Manipulation
        - Other Built-in Statements
    - Conditional Statements
        - The If Statement
        - The For Statement
        - The Include Statement
        - The Depends Statement
    - Summary
- Arrays
    - Construction
    - Indexing
    - Other features
- A PDriver Example
    - The Pi.pds Example

Using PDriver Tools

This guide serves as a tutorial for PDriver, the parametric Job Driver. The following tutorial describes PDriver and explains its use.

This section explains the basics behind using each of the PDriver tools, including `pdriver`, `bsub`, `bcoll`, `bstatus`, and `bcancel`. These commands are the basis of using PDriver, and are used in further sections to run and control PDS scripts.

PDriver tools are included in the GridServer SDK. For more information about installation, see the TIBCO GridServer® InstallationTIBCO GridServer® Installation.

Product Overview

PDriver, or the Parametric Job Driver, is a Driver that can execute command-line programs as a parallel processing service using the GridServer environment. This enables you to take a single program, run it on several Engines, and return the results to a central location, without writing a program in Java or C++.

PDriver achieves parallelism by running the same program on Engines several times with different parameters. A script is used to define how these parameters change. For example, a distributed search mechanism using the `grep` command can conduct a brute-force search of a network-attached file system, with each task in the Service being given a different directory or piece of the file system to search.

One way PDriver scripts can achieve parallelism is to iteratively change the value of variables that are passed to successive tasks as parameters. A script can step through a range of numbers and use each value as a parameter for each task that is created. Or, a variable can be defined containing a list of parameters.

PDriver uses its scripting language, called *PDS*, to define how jobs occur. These scripts can also be used to set options for a PDriver Service, such as remote logging and exit code checking.

Before You Begin

This guide assumes that you already have a Manager running and know the hostname, username, and password. If this isn't true, see TIBCO GridServer® Installation or contact your administrator.

You must also install the GridServer SDK, which includes PDriver and code examples. See TIBCO GridServer® Installation.

Running PDriver

There are multiple ways to start PDriver and run a script, depending on if you want to do so synchronously or asynchronously. We'll briefly explain each method, and then show how to run some sample code included with the GridServer SDK.

To start, open a command shell and change directories to the root level of the GridServer SDK, then change to the `pdriver` directory.

Running Synchronously

The simplest way to start PDriver is to run it synchronously. This runs through each block of the PDS script, waiting for each step to complete. This is the easiest way to run a test script, provided you don't mind waiting for the results.

For example, here is a very simple PDS script, let us call it `simplejob.pds`:

```
job simplejob
```

```
task 10

    log "hello, world"

end

end job
```

This PDS script contains a single `job` block, and nothing more. It creates ten tasks and each one writes a hello message to the Engine log.

Run this script by typing the following code:

```
pdriver simplejob.pds
```

This launches PDriver, sends the ten tasks to the grid, and waits for the results to return. The Driver log messages are printed on your console window, and you can see as each task is submitted to and received from the Engines running on your grid. The Driver logs are also captured in a timestamped file created in a `log` directory; this directory is created in the directory PDriver is invoked in.

This quick example doesn't do much, except for writing to a log. But a more complex script gives each Engine an executable to run, with different inputs for each task. More advanced scripting can possibly move data files or initialize databases before jobs or tasks are run, and results can be collected after each job or task.

For now, you can see the results of `simplejob` in the Engine log files. To do this:

1. In the Administration Tool, go to **Grid Components > Engines**> **Engine Admin**.
2. Select an Engine from the list.
3. Click the **Actions**link, and select **Log Files**. This opens a window containing a list of links to the Engine logs on that Engine.
4. Click a link to view the most recent log, and you'll see the "hello, world" message written by each task of the simple job.

Running Asynchronously

While it might be simple to run PDriver synchronously, waiting for the results to return might be impractical in a production scenario, especially if each task takes time to complete. An alternative

is to run the PDS script asynchronously, using the `bsub` flag. This runs the entire script except for post-job processing. Instead of waiting for tasks to complete, it immediately returns a batch ID, which can later be used to fetch results.

Let's modify the above `simplejob.pds` to add a `postjob` block with a hypothetical command. Here, it is another simple `log` command, but in a real job, it might be a command that copies, collates, or otherwise post-processes task results into a final result. Let us call this script `simplejob2.pds`:

```
job simplejob



task 10

    log "hello, world"

end

postjob

    log "job processed."

end

end job
```

Run this script by typing the following command:

```
pdriver -bsub simplejob2.pds
```

In the Driver logs displayed on the screen, there are lines that look similar to the following lines:

```
03/02 14:54:16:265 Info: Batch job has been submitted

03/02 14:54:16:265 Info: Collect outputs using batch id: 61924497789921878
```

The batch ID is important; use this to retrieve your results. In a production scenario, you can extract this ID from the Driver log file with grep or some similar approach.

After the job has been submitted, it runs on its own, and the results are queued. You can retrieve the results when you are ready by typing the following command:

```
pdriver -bcoll 61924497789921878 simplejob2.pds
```

Running Commands

Aside from running PDS scripts to iteratively run several instances of an executable with different inputs, it's also possible to run PDriver to run a single command on the Grid, and then collect the results later. This ca nbe used to run a one-time command, or to use another scripting language to run a multi-task job.

The `bsub` command can be used to run a command on a single Engine in the Grid. For example, if you have Windows Engines on your Grid, you can type the following command (UNIX users must replace the `dir` with `ls`.):

```
bsub dir
```

This submits a single-task job to an Engine, and returns the batch ID in the Driver logs.

The `bcoll` command is used to collect the results after using `bsub`. To collect the results, you simply need to give `bcoll` the batch ID. For example:

```
bcoll 149463216266848451
```

The results are fetched into a new directory with the same name as the batch ID. The directory contains a `bsub.err` file with the stderr output of the job, and a `bsub.out` file containing the stdout output of the job. In this case, the `bsub.out` file contains a directory listing of the Engine that ran the job.

Other PDriver Commands

In addition to `pdriver` and the `bsub` and `bcoll` commands, there are two other utilities you can use when running PDriver jobs.

First, the command `bstatus` returns information about job, batch, and Engine status. For example, type the following command:

```
bstatus -jobs
```

This displays the name, job IDs, tasks, priority, and status of all of the PDriver jobs recently submitted. You can also use the options `batches` and `engines` to display information about recently submitted batches and available Engines.

Another common task is stopping a submitted job. This is a two part task: removing pending jobs, and killing running jobs.

To remove a pending batch job, use the `bcancel` command, along with a batch ID. The batch ID is the same one returned when you initially ran `pdriver -bsub` or `bsub`. If you don't have the batch ID, you can look at the results of `bstatus -batches` to determine it.

The `bcancel` command removes jobs that are still waiting in the queue for execution, but it can't stop a job that has already started running and that now exists as one or more tasks on your grid.

To stop a running job, you can use another `bstatus` option called `canceljob`. But instead of the batch ID, you need to determine the Service ID of the job. This can be found by using `bstatus -jobs`, or by looking at the **Services > Services > Service Session Admin**page in the Administration Tool. To cancel a job, type the following command, where *serviceid* is the Service ID of the job:

```
bstatus -canceljob serviceid
```

Summary

- The `pdriver` command can run a PDS script synchronously.
- To run a PDS script asynchronously, use `pdriver -bsub`.
- Retrieve PDS script results with `pdriver -bcoll`.
- `bsub` and `bcoll` can be used to submit a single command and recall its results from the grid.
- Status on batches, jobs, and Engines can be displayed with `bstatus`.
- To kill a pending batch, use `bcancel` and a batch ID; to kill a running batch, user `bstatus -canceljob` and a Service ID.

PDS Structure and Variables

PDriver uses its scripting language, called *PDS*, to define how jobs occur. These scripts can also be used to set options for a PDriver Service, such as remote logging and exit code checking. This section describes the various parts of a typical PDS script and how to write a basic PDS script.

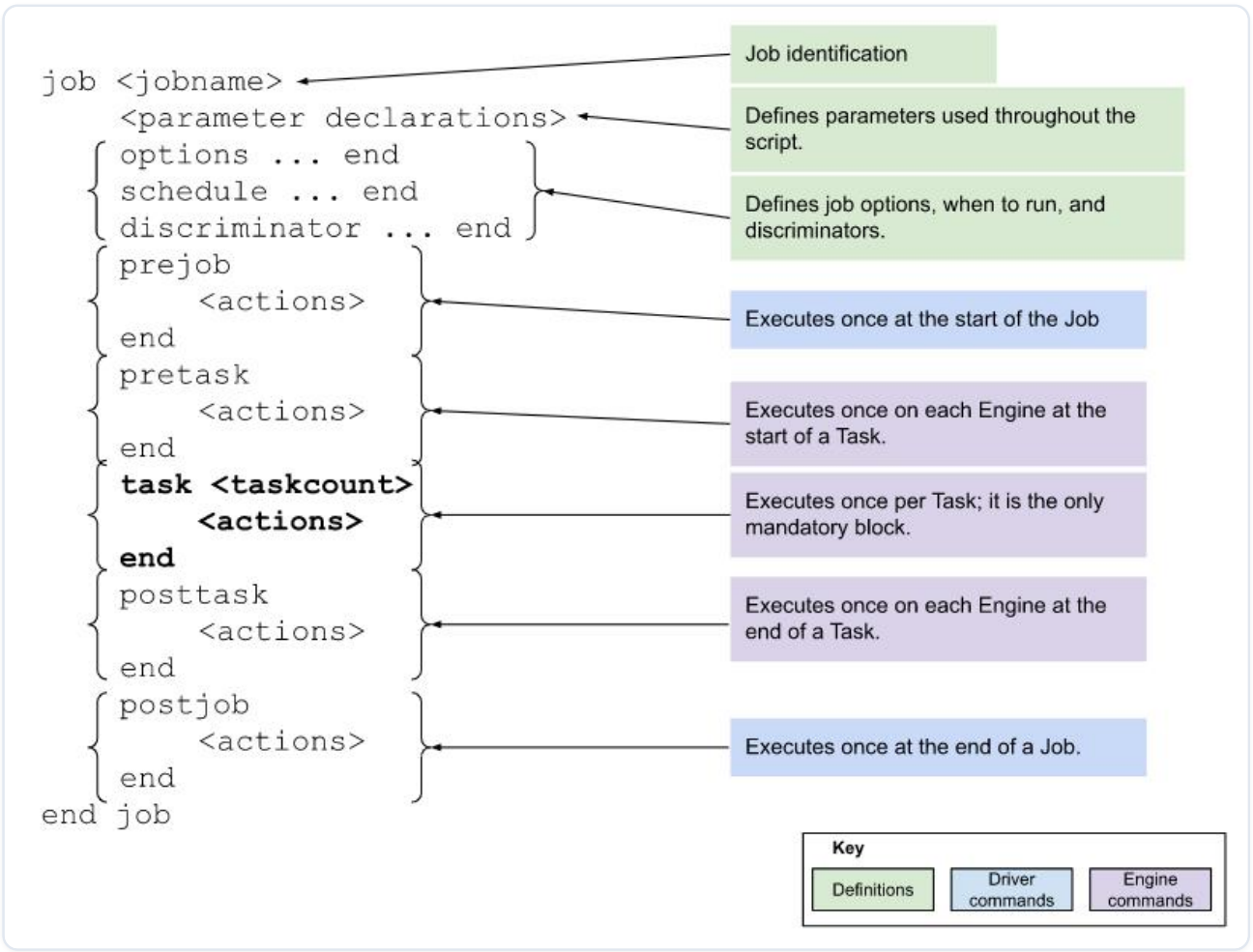
For a complete reference to the PDS scripting language, see the TIBCO GridServer® Developer's Guide.

PDS Structure

A PDS script typically coordinates three parts of a PDriver job:

1. Provide a pre-processing step to distribute inputs.
2. Provide a processing step to be done on Engines in parallel.
3. Provide a post-processing step to consolidate outputs.

To accomplish this, a PDS script is structured into different *blocks*, which are run at different points in the job's lifecycle. Each block can contain statements (or actions) or other specific options. This section explains what blocks are used in a basic script, and what the other blocks are for.



Structure of a PDS file

The Basic Script

All of the blocks in a PDS script are optional except two. Each PDS script requires the following blocks:

1. A basic PDS script consists of a `job` block. This begins with the line `job` *jobname*, where *jobname* is the name of your PDS job, and ends with `end job`. More advanced scripts can contain multiple `job` blocks which are run serially by default.
2. The other required block is the `task` block which is in the `job` block, and begins with `task` *taskcount* and ends with `end`.

Our `simplejob.pds` script from the previous section, contains these two required blocks:

```
#From the last section

job simplejob



task 10

    log "hello, world"

end

end job
```

A few other points to mention about PDS scripts. First, PDS is case-sensitive. Also, in PDS, whitespace is not significant. In fact, you can include multiple statements on one line, or even write an entire PDS script on a single line, although it's easier to read with a statement per line. And any text after a '#'sign on a line iss ignored, and can be used as a comment, as shown above.

Note that each of the remaining blocks we discuss typically begin with the name of the block and end simply with an `end`.

The Options Block

One of the additional blocks you can define in a PDS script is the `options` block. Within it, you can select set job options and descriptions, job priorities, and other settings.

```
options

    onerror ( fail | retry | ignore )

    maxFailedTaskRetries = val (default: 3)
```

```
    enableBlacklisting = true|false>

    jobPriority = val (default: 5)

    autoCancelMode = "always" | "never" | "libloadfailure"

    jobOption key val

    jobDescription key val

end
```

The Schedule Block

When submitting a job asynchronously with `bsub`, you can declare a start time using the `schedule` block, and also send an email when a Service is complete. This block has no effect if the job is submitted synchronously.

The `schedule` block can contain a `type` property, which defines what type of schedule is used. The possible values are relative, for defining a Service that starts a certain number of minutes after submission, and absolute, for a Service that starts at a defined time.

After inserting a type property, you must insert a time property to indicate the time when the Service must start. For relative services, use the `minuteDelay` property, set to an integer. For absolute properties, use the `startTime` property, set to a string in the `"mm/dd/yy hh:mm AM|PM"` format.

Allowed schedule declarations are:

Relative

```
schedule

    type = relative

    minuteDelay = val

end
```

Absolute

```
schedule

    type = absolute

    startTime = "mm/dd/yy hh:mm AM|PM"

end
```

With either type, declaring `email="string"` in the `schedule` block sends an email to the address given in the string when the Service is complete.

The Discriminator Block

Discrimination is a feature of GridServer that allows you to selectively use Engines based on their properties. They enable you to specify if a job or a task can be taken by certain Engines. Normally, discriminators are attached to Services using the GridServer API. However, with PDriver, you can specify job-level or task-level discriminators in your PDS script.

In the `discriminator` block of a PDS script, you can declare discriminators, which consist of a property name, a comparator, and a value:

```
property-name ==|!=|<|>|<=|>= param-value
```

For example, putting the following code in a `discriminator` block discriminates against Engines with a value of 350 or less in the `cpuMHz` Engine property:

```
cpuMHz > 350
```

Declaring a discriminator creates a job-level discriminator by default. You can also create a discriminator that requires conditions to be met before the discriminator is applied to a task. This is accomplished by using an `affects` clause in the `discriminator` block.

Use of the `affects` clause specifies conditions that must be met for the discriminator to be applied to a particular task. For example, the following discriminator block shows how to apply the same discriminator as above, but only for the first five tasks:

```
discriminator

    affects

        $DSTASKID <= 5

      end

    cpuMHz > 350

end
```

Multiple discriminator blocks with `affects` clauses can be used.

The property name refers to Engine properties. A predefined set of properties are assigned to all Engines by default. You can assign additional properties to Engines by selecting **Edit/View Properties**from the Actions list in **Grid Components > Engines > Daemon Admin**in the Administration Tool.

Prejob and Pretask Blocks

PDriver Jobs and tasks might require preprocessing. For example, you might need to prepare files, move them to an Engine, start a database, or do other preparation before the `task` block begins. To provide this functionality, two blocks are available to run before the task block: `prejob` and `pretask`. The two blocks start with either `prejob` or `pretask` and end with an `end`, and they can contain any number of statements. Also, both blocks of code must be located before the `task` block in your PDS script.

One difference between these two blocks is that `prejob` runs once, at the start of the job, and `pretask` runs once per Engine, prior to the start of the first task to run on that Engine. Also, the `prejob` block code executes on the Driver (that is, the machine on which you are running PDriver), and the `pretask` block code executes on the Engine.

```
prejob

    statements

end

pretask
```

```
    statements

end

task taskcount

    statements

end
```

Posttask and postjob blocks

After a PDriver task or job runs, you might want to run code to do some postprocessing. For example, you can use `posttask` blocks to clean up the state on the Engine. You can use the `postjob` blocks to move results to other locations. Like the `pretask` and `prejob` blocks, simply surround your code with a `posttask` or `postjob` keyword at the beginning, and an end keyword at the end. Locate both blocks of code after the task block in your PDS script.

Code in the `posttask` block runs on each Engine after the job completes, while code in the `postjob` block runs once on the Driver. One important difference in the `posttask` and `postjob` blocks is when they are executed. The `posttask` block happens after all of the tasks are completed. There's no guarantee that they happen immediately after the tasks are done, but they finish before the job ends. If you are running your job synchronously, the `postjob` block runs when the job finishes. And if you used `pdriver -bsub` to submit a job, the `postjob` block runs when you collect the results.

Variables and Parameters

There are several items implemented in PDS to make it easy to manipulate variables and use them as parameters. This section explains the basics.

The Basics

PDS has two primitive types: strings and floating-point numbers. You don't need to declare variables, and they can change types. Variables are referenced by preceding them with a dollar sign. Values are converted to the appropriate type depending on context. For instance, a string is converted to a number when it appears in an arithmetic expression. The grammar forbids certain combinations of expressions to catch common mistakes.

For example, if you add a string to a number, the string is converted to a number.

```
a = 7

b = "1.5"

c = $a + $b                 # value is 8.5

d = 7 + "1.5"               # disallowed
```

You can also do variable substitution within quotes. Use braces around a variable to separate it from adjacent text. For example:

```
a = "New"

b = "$a York"                # b equals "New York"

c = "${a}ark"                # c equals "Newark"
```

About Scope

You can assign variables within the `job` block or within any other block, and they are local to that block. For example, if you assign a variable in the job block but outside and before any other block, you can use it in the `prejob`, `postjob`, `pretask`, or `posttask` blocks. Inside a block, a variable can be assigned as global variable, which is visible within other related blocks, with the following syntax:

```
    global a = 5.2
```

Summary

- A PDS script typically coordinates three parts of a PDriver job: provide a pre-processing step to distribute inputs; provide a processing step to be done on Engines in parallel; and provide a post-processing step to consolidate outputs.
- A PDS script is structured into different blocks, which are run at different points in the job's lifecycle.
- A basic PDS script consists of a `job` block.
- The other required block is `task` block.

PDS Statements

Statements are used within a PDS script to execute commands, manipulate files, log messages, and perform other tasks that you might need to do during a job. They can also be used to add more advanced control to a PDS script, with statements like `if`, `for`, and `depends`.

This section describes common statements used in PDS scripts.

Built-in statements

Built-in statements are OS-independent commands that can be used inside any block in a PDS script. There are a few exceptions, however. You cannot use `onerror`, and `throw` in every block. For more information, see TIBCO GridServer® Developer's Guide.

Arguments to all statements except for `onerror` must be enclosed in quotes. Quoted arguments might contain line breaks only if they are escaped by backslashes.

Command Execution

To execute a command on a local machine within a PDS script, you can use the `execute` statement. A subshell is spawned for the command, unless specified otherwise with the shell command. The subshell has the path available on the Engine or Driver. The execute statement also enables you to set the stdin, stdout, and stderr used by the command. For example, the following script is a simple PDS script that uses `execute`:

```
job simplecommand

task 1

    execute stdin="input.txt" stdout="output.txt"

    stderr="error.txt"

        "sort"

end

end job
```

This script creates one task, which takes the contents of `input.txt`, sends them to the `sort` command on an Engine, collects the results into the file `output.txt`, and collects any errors in `error.txt`. All three files are located on the Engine in the Engine's installation directory.

You can also use the “backtick” syntax (such as `‘command‘`) to do the same thing as `execute`, but the input, output, error, and shell cannot be set.

The `shell` statement can be used to specify the shell to be used for commands run by the `execute` command. The default is `/bin/bash` for UNIX and `cmd.exe` for Windows. This can be set to `none`, whereupon no shell is spawned.

You can also change the behavior of a script based on the return code of a command run with the `execute` statement by using the `onerror` statement. For more information about the `onerror` statement, see the TIBCO GridServer® Developer's Guide.

File Manipulation

Several built-in statements enable you to manipulate files and directories on the Driver or Engines. These can be used to move input and output files to and from the Engine.

The `copy` statement copies a file. Typically, the `prejob` block is used to copy input files from the Driver machine to the staging directory on the Manager, which can be referenced as `$DSSTAGEDIR`. In the `task` or `pretask` block, one can then copy that file from the staging directory to the Engine’s local filesystem. Output files can be copied in the reverse direction.

You can remove files from the Driver or Engine using the `delete` statement. The wildcard `*` is supported.

Similarly, the `mkdir` and `rmdir` statements create or remove directories. To use `rmdir`, the directory must be empty.

The file manipulation statements are not OS-dependent and pathnames are automatically translated to work on the appropriate platform. For example, `mkdir "sample/log1"` creates `sample\log1` on Windows systems.

The following script uses files for standard input, output, and errors. It essentially copies everything from the input file to the output file.

```
job simplefile

prejob

    copy "hello.txt" $DSSTAGEDIR

end
```

```
task 1

    copy "$DSSTAGEDIR/hello.txt" $DSWORKDIR

    execute stdin="$DSWORKDIR/hello.txt" stdout="$DSWORKDIR/output.txt"
stderr="error.txt"

        "sort"

    copy "$DSWORKDIR/output.txt" $DSSTAGEDIR

end

end job
```

Other Built-in Statements

Several more built-in statements are available. The following table contains statements that might be useful in your PDS scripts.

STATEMENT	DESCRIPTION
`log "` *log-message* `"`	Writes a message to the Manager log (in prejob or postjob blocks) or the Engine log (in pretask, task, or posttask blocks.)
`env (variable)`	Returns the value of the specified environment variable. If the variable is a string literal, it must be enclosed in quotes. An empty string is returned for nonexistent variables
`throw "` *message* `"`	Causes the task to fail, and throws an exception. The message is displayed on the Driver and written into the Driver log. This cannot be used in the posttask block.

Conditional Statements

Several conditional statements can be used within blocks to control the flow of a PDS script.

The If Statement

The syntax of the basic if statement is

```
if expression comparisonOperator expression then

    statements

end
```

PDS also supports `elsif` and `else` clauses. The `if` statement can be used inside a block (`prejob` , `pretask` , `task` , `posttask` , `postjob`), to conditionally execute statements. One typical use is to execute different commands depending on the Engine's operating system:

```
if $DSOS == "win32" then

    cmd = "dir"

else

    cmd = "ls"

end

execute "$cmd"
```

An `if` statement can also appear at the top level of a PDS script, to include or exclude global assignments or entire blocks. An example is

```
if $1 == "alwaysCancel" then

    options

        autoCancelMode "always"

    end

end
```

The `if` statement is not legal in the `options` , `schedule` , or `discriminator` blocks.

The For Statement

Although explicit looping over an array is not often needed, it can be achieved with the `for` statement:

```
for variable in expression

    statements

end
```

The expression must evaluate to an array. The variable takes on successive elements of the array on each iteration of the loop. Assignment to the loop variable is legal, but has an effect only for the remainder of that iteration. It does not alter the array.

The Include Statement

You can include another PDS script within a PDS script by using the `include` statement. For example:

```
include "filename"
```

This includes a PDS script contained in `filename`. If the filename is relative, it is interpreted as relative to the working directory from which you run PDriver. The PDS file must contain a full job, and can be used within any block.

The Depends Statement

Multiple `job` blocks can be included in a single PDS file, and by default, they run sequentially. It is also possible to define jobs that run other jobs that require the completion or failure of the current job using the `depends` statement. `depends` is used within a job block, and defines if another job after the current job in the PDS file must run either on the completion or failure of the current job. For example, the following code within a job runs `firstjob` if the current job succeeds, `secondjob` if it fails, and `thirdjob` in either case:

```
depends

    firstjob succeeds

    secondjob fails
```

```
    thirdjob succeeds or fails

end
```

You can also run the `pdriver` command with the `-parallel` flag to run multiple jobs in one PDS file in parallel.

Summary

- Built-in statements are OS-independent commands that can be used inside almost any block in a PDS script.
- Conditional statements can be used within blocks to control the flow of a PDS script.

Arrays

Arrays are fundamental to achieving parametric parallelism in PDS, because an array variable is implicitly indexed in the task block.

Construction

Arrays of primitive values can be constructed in several ways. A literal value is written as follows:

```
    a = [1, 2, 3, 4, 5]
```

Arrays can also be constructed by autorange expressions. The expression

```
    begin n end m step k
```

constructs an array starting with *n* and proceeding in increments of *k* until *m* is reached. More precisely, it constructs

```
    [n, n+k, n+2k, ..., n+rk]
```

where *r* is the largest integer such that *n+rk* <= *m*.

The expression

```
    begin n count c step k
```

constructs an array of *c* elements beginning with *n* and proceeding in increments of *k*, that is,

```
    [n, n+k, n+2k, ..., n+(c-1)k].
```

For example,

```
    begin 10 end 15 step 2
```

and

```
    begin 10 count 3 step 2
```

both construct the array:

```
    [10, 12, 14].
```

The third way to construct an array is to use the `split` function, which divides a string into array elements at whitespace. Quoted elements keep embedded whitespace and strip the quotes upon placement into the array. For example, on UNIX machines:

```
    split(‘ls‘)
```

returns an array of the files in the current directory.

Indexing

In most contexts, when a variable containing an array is referenced, the first element is returned. However, in the task block, the element corresponding to the ID of the currently running task is returned. (If the task ID exceeds the array size, the last element of the array is returned.) This feature makes it easy to write the most common kinds of distributed computations. For example, we can set up an array of values and run a command on each one in parallel with the following PDS script:

```
args = begin 100 end 200 step 5

task sizeof($args)

    execute "doit $args"
```

```
end
```

The `$args` in the execute statement expands to 100 for the first task, 105 for the second, and so on.

Explicit array indexing is not supported: the only ways to obtain an array element other than the first are implicit indexing in the task block, and the `for` statement. Even in those contexts, assignment to the variable holding the array changes the entire variable value, not the current element.

Other features

An array that includes both string and numeric values is legal. Arrays of arrays are not allowed.

The number of elements in an array can be determined by the `sizeof` function, as shown in the argument to the `task` block in the above example.

A PDriver Example

This section shows an example PDS script that calculates pi by using a distributed Monte Carlo calculation.

The Pi.pds Example

This example works by running an executable that can be built from `[Gridserver SDK dir]\examples\pdriver`. Build and deploy the executable using the supplied script.

A single script, `pi.pds`, runs several tasks, each with a different seed and multiple iterations.

The beginning of `pi.pds` starts the job block by defining the number of tasks and an output directory for the tasks.

```
# pi.pds

#

# performs a distributed monte carlo pi calculation.

#

job pijob
```

```
tasks = 50

outputdir = "pijob.example.out"

seed = begin 200 count $tasks step 3

iterations = 500000
```

Next, a task block must be written to run the executable. However, the task must run one of two different binaries, depending on the operating system of the Engine. A conditional is used to check the status of the `$DSOS` variable. If it is win32, the `PdriverPiCalc.exe` binary is run, otherwise, `pdriverPicalc` is run from the `$DSOS` resources directory. The program takes two parameters: the seed, and the number of iterations. The standard output and standard error from the program is redirected to the work directory. After each task is completed, the output and error output is copied back to the staging directory.

```
task $tasks

    onerror retry

    shell "none"

    if ( $DSOS == "win32" ) then

        execute stdout="$DSWORKDIR/pijob.$DSTASKID.out"

                stderr="$DSWORKDIR/error.$DSTASKID"

                "${gridLibraryDirWin}\commands\PdriverPiCalc.exe $seed
$iterations"

    else

        execute stdout="$DSWORKDIR/pijob.$DSTASKID.out"

                stderr="$DSWORKDIR/error.$DSTASKID"

                "${gridLibraryDirUnix}/$DSOS/commands/PdriverPiCalc $seed
$iterations"

    end
```

```
    copy "$DSWORKDIR/pijob.$DSTASKID.out" "$DSSTAGEDIR"

    copy "$DSWORKDIR/error.$DSTASKID" "$DSSTAGEDIR"

end
```

After the task block has run 50 times, the postjob block is run. It processes the output of each task with a `pi-average-win32.bat` or `pi-average-unix.sh` script.

```
postjob

    onerror ignore

    mkdir "$outputdir"

    copy "$DSSTAGEDIR/pijob.*" "$outputdir"

    if ( $DSOS == "win32" ) then

        execute stdout="PIVALUE.OUT"

                """%DS_SDK_DIR%\examples\pdriver\pi-average-win32.bat""
$outputdir"

    else

        execute stdout="PIVALUE.OUT"

                "$$DS_SDK_DIR/examples/pdriver/pi-average-unix.sh
$outputdir $tasks"

    end

end

end job
```

Speedlink Guide

- Introduction
  - Before You Begin
- Installing SpeedLink
  - Requirements
  - Installing SpeedLink
- SpeedLink Architecture
  - Overview
    - Use Cases
  - SpeedLink Architecture
    - Operations
    - Clients
    - Services
    - Administration
- Developing Applications for SpeedLink
  - Running a Service by Using SpeedLink
  - Service Options
  - GridServer and SpeedLink Service Differences
    - Improvements
    - Limitations
    - Not Supported
- Administration for SpeedLink
  - Monitoring and Reporting
    - Enabling and Configuring TaskInfo Reporting

    - Performance Tuning
        - Operation Queue
        - Allocation Monitor
        - Operation Dispatcher
    - Failure Modes in SpeedLink
        - Network Failure
        - Broker Failure
        - Engine Failure
        - Operation Failure
- SpeedLink Service Options
    - SpeedLink Service Options
    - Standard Service Options in SpeedLink

Introduction

The TIBCO GridServer® Administration is for administrators who maintain TIBCO GridServer® installations. It describes how TIBCO GridServer® works and how to use the GridServer Administration Tool. Topics include how to schedule and route Services, deploy resources, manage failover Brokers, and perform other frequent tasks. This guide also provides advanced information about security, tuning, database administration, and log files.

Before You Begin

This guide assumes that you know TIBCO GridServer® concepts. If you do not, see the TIBCO GridServer® Introducing TIBCO GridServer® guide for information about the TIBCO GridServer® component architecture and principles of operation.

Before beginning, you must already have a TIBCO GridServer® Manager running and know the hostname, username, and password. If this isn't true, see TIBCO GridServer® Installation or contact your administrator.

Installing SpeedLink

This section covers SpeedLink's system requirements and installation process.

Requirements

For current GridServer requirements see the GridServer Readme.

SpeedLink supports all Services except Command Services which do not benefit from lower latencies, and R Services. All other Services are supported including Java, .NET (`String/byte[]` only), and C++ —all compilers that are supported by standard Services.

Installing SpeedLink

SpeedLink is preinstalled in GridServer. To use SpeedLink, install the GridServer SDK the same way you install it to use other Services.

For more information about installing the SDK, see the TIBCO GridServer® Developer's Guide.

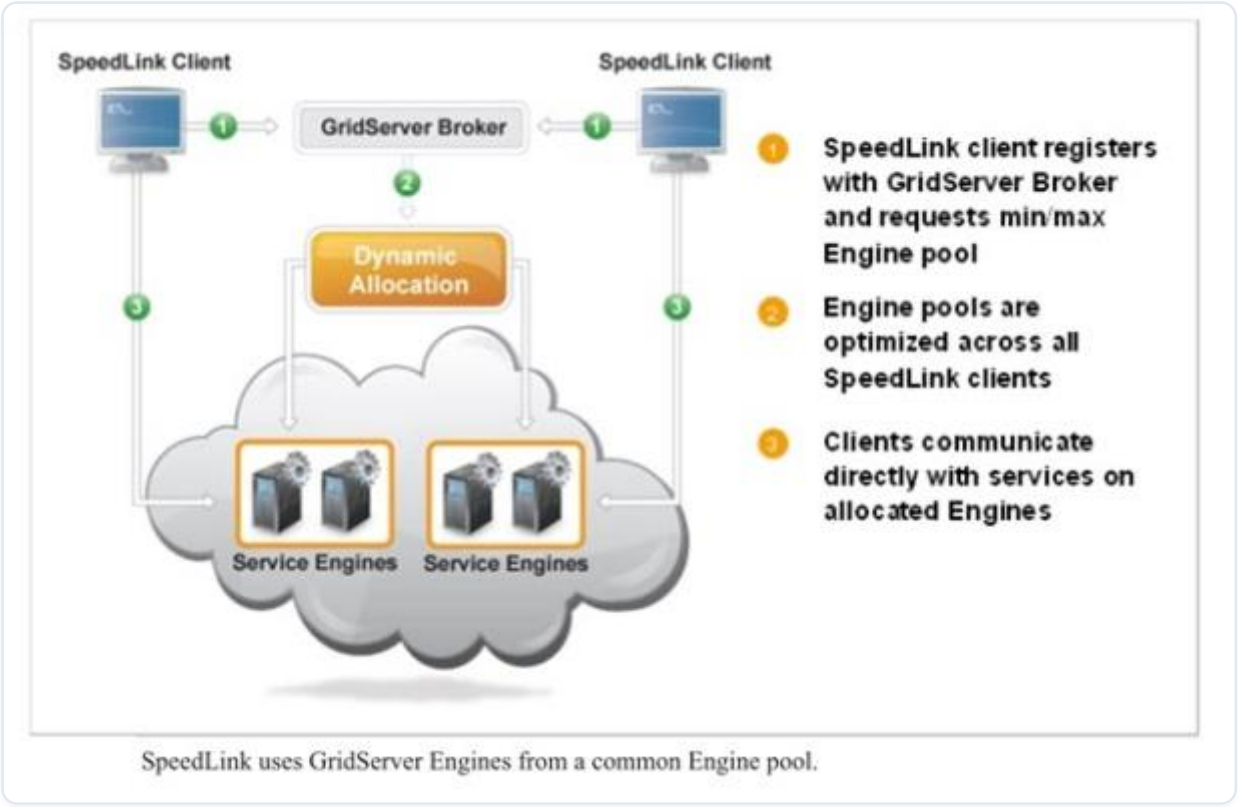
SpeedLink Architecture

This section covers SpeedLink architecture and describes appropriate uses for SpeedLink Services.

Overview

SpeedLink is a high throughput, low latency compute paradigm implemented using GridServer. SpeedLink can deliver end-to-end compute latencies of less than one millisecond, depending on the server and network environment, with no theoretical upper limit on aggregate grid-wide operation throughput. Individual clients can encounter a practical throughput limit based on the client system's performance.

SpeedLink uses the Engine pool of a GridServer installation, sharing Engines seamlessly with other GridServer Services. SpeedLink maintains key GridServer features like Service prioritization and Engine balancing, and standard Services can co-exist with SpeedLink Services, even within the same client or Engine process. SpeedLink Services request Engines from a GridServer Broker, and this allocation operates much like a standard GridServer Service invocation. Unlike standard Services, allocated Engines communicate directly with the SpeedLink client, resulting in the desired throughput and latency characteristics. Allocated Engines are periodically returned to the Engine pool.



SpeedLink uses GridServer Engines from a common Engine pool.

SpeedLink's design supports easy code migration from standard GridServer Services to SpeedLink Services, and back again to standard Services if desired. The SpeedLink API provides the `Service` interface for SpeedLink Services. The application programmer simply calls the `createService` method on a different factory class. The resulting `Service` interface (or virtual object, depending on the chosen implementation language) can be used the same way you use a standard Service.

Use Cases

The canonical use case for standard GridServer Services is risk analysis. Risk is highly computational, with typical run times in the order of seconds or minutes, and it is infinitely hungry for compute nodes, so horizontal scalability is preferred over low latency.

In contrast, a good example of a suitable SpeedLink Service is a Forex trading application, where portfolios, trades, and strategies need to be constantly re-evaluated based on market conditions delivered from a Forex market data feed. This use requires high throughput since Forex rates might change dozens of times a second, and there are hundreds of positions being evaluated. At the same time, the computational requirements for individual positions or strategies are low, so low latency is preferred over horizontal scalability.

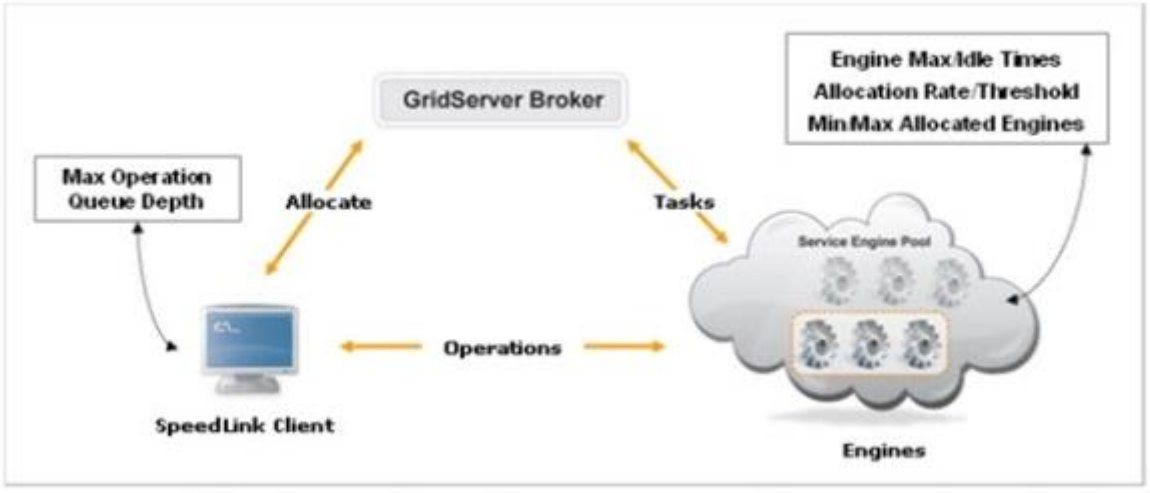
SpeedLink Architecture

Like GridServer, SpeedLink uses a client application that calls Services running on many Engines, and then collects and handles the results. But instead of using the Broker as a middleman for all

communication, SpeedLink clients only communicate with the Broker when reserving those Engines. Submitted or executed invocations are then treated as low-latency operations and directly communicated to Service code running on GridServer Engines. SpeedLink enables high throughput by bypassing the Broker.

Operations

In GridServer, the term *task* describes the execution of a Service invocation, including reserving an Engine, along with its input and output. For SpeedLink Services, the SpeedLink task executes many invocations over the duration of the Engine reservation. Because of this, we introduce the term *Operation* to refer to each invocation execution. The are many operations within one task.



SpeedLink architecture: Clients and Engines communicate with Broker; operations handled directly.

Clients

The SpeedLink client is the program developed to call a Service implementation running on multiple Engines, and then collate or process the operation results. Your application code in the SpeedLink client creates its Services through the `SpeedLinkServiceFactory`, setting additional options as appropriate. The factory's `createService` method returns a concrete class that implements the Service interface. Internal to the concrete class, `submit`, `execute`, and a subset of other Service methods are implemented. Service Proxies are not supported; client code must use the `Service.submit` or `execute` methods to invoke Service code.

The direct connection between a SpeedLink client and a SpeedLink Engine is initiated from the Engine back to a server socket on the client. Each client requires only one open port that shared between all Service instances created.

Services

The SpeedLink Service implementation, running on the GridServer Engine, corresponds to the SpeedLink client implementation. SpeedLink clients connect to the SpeedLink Service implementation. Like GridServer, SpeedLink operations have the following lifecycle:

1. Your application client tells the SpeedLink client what application Service to use.
2. The SpeedLink client passes that information to the SpeedLink Service implementation.
3. The SpeedLink Service implementation manages the application Service.

Consequently, application Service code need not change to run as a SpeedLink Service. If you deploy a Service with low-latency performance on your GridServer Broker, you can access it as a standard Service by having your client use `ServiceFactory`, or access it as a SpeedLink Service by using `SpeedLinkServiceFactory`. You can even access the same Service code as both standard and SpeedLink Service instances from within the same client process.

In contrast to GridServer Services, the SpeedLink Service itself manages the connection back to the SpeedLink client, receiving Service input and returning output, like a simplified version of the GridServer Engine's connection to a Broker. It also watches the `ALLOCATION_TIME` and `IDLE_LINGER` timers, ending the Engine's reservation when those total or idle timers expire, respectively.

Administration

Administration and maintenance of SpeedLink Services is almost identical to GridServer because SpeedLink uses GridServer Brokers and Engines to maintain an Engine pool. SpeedLink has its own Service Type; SpeedLink clients are managed on the Client Admin page, and SpeedLink Services appear on the Dashboard, as do GridServer Services.

However, there are features not supported in SpeedLink that might change some aspects of your grid management. These include:

- Reporting— due to the large amount of reporting information generated, SpeedLink's task-level reporting is summary only; operation-level reporting must be performed by the client.
- Service Options— for example, a grid configured to use redundant scheduling, which is not supported with SpeedLink, might need to be managed differently.

Despite these differences, an administrator already familiar with GridServer's Administration Tool has no difficulty managing SpeedLink Services.

Developing Applications for SpeedLink

A Service object on a client can create and use a Service implemented in the same or another language. This section discusses how to implement a SpeedLink Service client—the code calling a Service on the Driver, and a SpeedLink Service—the code that runs on the Engine).

Running a Service by Using SpeedLink

Almost any GridServer Service can be executed by using SpeedLink. You can use an existing `ServiceType` and its Grid Libraries. All you need to do to run a Service by using SpeedLink is, in the client code, replace the `ServiceFactory` class with `SpeedLinkServiceFactory`, and `ServiceInvocationHandler` with `SpeedLinkInvocationHandler`.

The factory class returns an object of type `SpeedLinkService`, which extends the `Service` interface. The most important feature of the new type is that it enables access to detailed reporting information. If you are not interested in this information but might to have interchangeability with standard Services, you can instead use the `Service` interface. This factory class also has all additional options needed for the Service declared as string constants.

This example creates a factory object and synchronously calls a Service implementation in Java:

```
// Create a Service instance of JavaAdderExample

Service s =

    SpeedLinkServiceFactory.getInstance().createService(

        "JavaAdderExample", initData, options, description,

        condition);

.........

// Perform Synchronous add

Object[] arguments = new Object[] { new Double(5), new Double(6) };

Double sum = (Double)s.execute("add", arguments);

System.out.println("Result of add: " + sum);
```

Service Options

SpeedLink support many of the existing Service Options, and has additional options for tuning specific to SpeedLink. For a complete list of all available SpeedLink options and any differences for GridServer options with SpeedLink see SpeedLink Service Options. For an explanation of how to set Service Options to tune performance, see the TIBCO GridServer® Administration.

Service Options can be set in two ways: in the GridServer Administration Tool in the Service Type Registry page, or when creating the Service Session with the client. If an option is set in the registry, the client cannot override it. If it is left as `[not set]` in the registry, and the client does not set it, the default value is used. When an option is not available for SpeedLink, a warning message is logged.

GridServer and SpeedLink Service Differences

The following sections detail the differences between standard GridServer Services and SpeedLink Services.

Improvements

The following items are improved in SpeedLink Services versus GridServer Services:

- Typical invocation latency on a generic 100BaseTX network is around one millisecond for SpeedLink and around 25 milliseconds for standard Services.
- Typical maximum client throughput rates are 1,000 invocations per second for SpeedLink and 100 invocations per second for standard Services.
- SpeedLink throughput is limited by client performance, not Broker performance, maximum Broker-wide throughput is limited to about 1,000 invocations per second for standard Services, but could easily scale to 100,000 invocations per second for SpeedLink Services.

Limitations

The following features are limited in SpeedLink Services:

- SpeedLink's higher throughput means that task-level reporting is limited to summary information in the reporting database. Fine-grained operation-level reporting can be performed via the SpeedLink API on the client side.
- SpeedLink's horizontal scalability—the number of Engines per Service— might be limited to hundreds of Engines per client, whereas standard Services are can scale over 5,000 Engines per Service.

- SpeedLink limits the data size returned from an Engine to 1 MB, including the serialization overhead. Returning larger amounts of data significantly reduces SpeedLink's performance.

Not Supported

The following features are not currently supported in SpeedLink:

- Automatic rescheduling of failed Service operations—any failed operation is immediately reported to the application as an invocation failure exception. This is primarily because there is no optimization advantage to having the SpeedLink code retry versus customer code.
- Redundant scheduling, since it has no benefit in low latency applications.
- Data encryption, data compression, or SSL.
- State update.
- Invocation cancellation.
- Task resubmission.
- Operation-level discrimination.
- Deferred collection such as `COLLECT_LATER` and `COLLECT_NEVER`.
- Proxies are not supported; client code must use the `Service.submit` or `execute` methods to invoke Service code.
- SpeedLink does not support the XML Serialization mode, which is typically only used when you require cross-language use of objects, or when a client does not have access to the Service implementation classes.

Administration for SpeedLink

Administering a GridServer grid using SpeedLink Services is essentially identical to managing a grid running standard Services. Knowing the differences can be helpful when running or your grid. This section discusses issues unique to SpeedLink. Refer to the GridServer documentation, and specifically the *TIBCO GridServer® Administration* for more details.

Monitoring and Reporting

SpeedLink Services, like standard Services, are viewable on the GridServer dashboard and Broker console. On the **Services > Services > Service Session Admin**page, there is a Service Type Name column; this is the **SpeedLink**for these Services.

The biggest difference between standard Services and SpeedLink Services is in the granularity of administration and reporting. Since throughput levels can be much higher on SpeedLink Services

(it's designed to a nominal 10,000 operations per second per Broker target), it is counter-productive to attempt to report every operation detail to the administration console or reporting database. Instead, Engine reservations are presented as tasks, with operation summary reports presented in the task information column.

Enabling and Configuring TaskInfo Reporting

By default, Task Event reporting is not enabled. To enable reporting, set the value of the SpeedLink Service Option `SUMMARY_INFO_ENABLED` to `true`. Once Task Event reporting is enabled, summary operation information is stored in the TaskInfo field of the reporting database after the Service completes. The following summary information is recorded:

- Number of completed operations
- Number of failed operations
- Service allocation time for task—specified in milliseconds

The format of the database entry is determined by the value of the following SpeedLink Service Option: `SUMMARY_INFO_FORMAT`. The default value is: `operations={0}, failures={1}, allocationTime={2}`. You can change the value of this option; for more information about the available message format options consult the Javadoc for `java.text.MessageFormat`.

Set the properties described here in the `createService` method of your `SpeedLinkServiceFactory`. You can set reporting and the reporting format independently for each SpeedLink Service.

Performance Tuning

There are three main components to the SpeedLink client implementation that can be configured with Service Options depending on your application, hardware, and network: the Operation Queue, the Allocation Monitor, and the Operation Dispatcher.

Operation Queue

The operation queue contains the low-latency operations that the client needs to have executed on the Grid. When an operation is submitted, it is placed on a queue from which another thread sends the operation to any idle Engine.

The operation queue size is controlled by the `MAX_QUEUED_OPERATIONS` option. If the queue of pending operations not yet submitted to Engines reaches this size, any further submissions cause the client to block until room is available in the queue.

Allocation Monitor

The allocation monitor watches the queue and the number of allocated Engines, and requests Engines to be reserved based on four options.

The minimum and maximum amount of Engines that can be allowed to work concurrently on a Session are specified upon session creation, as `MAX_ALLOCATED_ENGINES`, `MIN_ALLOCATED_ENGINES`. This essentially specifies the number of active reservation tasks at any given time. There are no specific guarantees that Engines are available to take those tasks, however, a session can be given `URGENT` priority to provide for some level of guarantee of availability. The minimum specifies how many idle tasks are maintained if the operation queue is empty; the maximum caps the amount of tasks regardless of the queue.

The size of the operation queue is defined by `ALLOCATION_OPERATION_QUEUE_THRESHOLD`. Once the queue rises above the queue threshold the allocation monitor starts requesting additional Engines at the rate specified in the `ALLOCATION_RATE` option, and continues requesting at that rate until the queue size drops below the threshold, or until the maximum reservation count is reached. The queue monitor might also automatically extend a new reservation when an Engine completes its reservation after reaching one of its timeouts.

If SpeedLink tasks were allowed to remain active indefinitely, then this session could starve the grid of Engines, as this essentially becomes a session with long-running tasks. To prevent this, the `ALLOCATION_TIME` option specifies the maximum amount of time a task can be performing operations. If an operation is in progress when it times out, the operation is allowed to complete first. Also, the SpeedLink implementation adds “noise” to the timeout value so task completion events become randomly staggered. Additionally, the `IDLE_LINGER` time specifies how long a task can remain idle without performing operations before it automatically deallocates itself.

Operation Dispatcher

The operation dispatcher handles sending operations to Engines and correlating results with the original requests. Because all Engines have already been reserved for the Service, prioritization is not necessary; operations are simply assigned to any available Engine. Operation-level conditions are not supported. It tracks the `OPERATION_TIMEOUT` so that an operation that hangs can be stopped.

Failure Modes in SpeedLink

As with GridServer Services, there are several possible points of failure with a SpeedLink Service. This section describes the various issues that can cause a Service to fail, and how they are handled.

Network Failure

From the client point of view, there are two types of failure related to the network and connections between components:

- Socket close, connection failure, and other interruption errors.
- Socket blocks on a read until timeout as defined by the `OPERATION_TIMEOUT` option.

In either case, on failure the client automatically retries an operation up to a total of three times.

Broker Failure

If a client is disconnected from the Broker and the connection times out, the client resubmits outstanding reservation tasks to the new Broker as soon as it can log in.

It is possible that this is due to the Client-Broker connection broken, but the Broker is not down and Engines are still connected and running tasks.

If the Driver timeout is set the same on the Broker and the Driver, the Tasks on the original Broker are canceled, so the existing connections are closed and the Service fully recovers.

Engine Failure

Engine failures generally appear as a network failure from the client when it can't maintain a connection to the Engine. When the Service fails, it is retried on another Engine as if it had timed out.

Operation Failure

If an operation fails, the failure is propagated back to the handler, as it is in standard Services. However, SpeedLink does not provide an auto-retry, like GridServer. (Because there is no Broker middleman, an automatic retry is identical to the client code resubmitting the operation, so it provides no added benefit.)

When an operation fails, there might be some cases in which you might want the reservation task to continue to process operations; in other cases you might also want the task to also fail, such as when using Blacklisting to prevent that Engine from processing any subsequent operations. The Boolean option `TASK_FAIL_ON_OPERATION_FAILURE` sets this option.

SpeedLink Service Options

The tables in this section describe SpeedLink Service options and standard Service options and their use in a SpeedLink Service.

SpeedLink Service Options

The following table lists all SpeedLink Service Options. These options are also further described in the API documentation included in the SpeedLink installation.

SERVICE OPTION	DESCRIPTION
`SUMMARY_INFO_ENABLED`	When `True`, summary Task Event reporting is enabled for the Service. Default is `False`.
`SUMMARY_INFO_FORMAT`	The default value is: `operations={0}, failures={1}, allocationTime={2}`. For more information about the available message format options consult the Javadoc for `java.text.MessageFormat`.
`ALLOCATION_OPERATION_QUEUE_THRESHOLD`	The operation queue threshold at which more Engines are allocated to this session.
`ALLOCATION_RATE`	The rate in Engines/second, at which Engines are allocated to this session when below the `MAX_ALLOCATED_ENGINES` and above the `ALLOCATION_OPERATION_QUEUE_THRESHOLD`.
`ALLOCATION_TIME`	The amount of time, in milliseconds, that an Engine can remain allocated.
`IDLE_LINGER`	The amount of time, in milliseconds, an allocated Engine remains idle before it deallocates itself.
`MAX_ALLOCATED_ENGINES`	The maximum number of Engines that are allocated to this session at any time.
`MAX_QUEUED_OPERATIONS`	The maximum total number of operations from this Service Session that can be queued on the client.
`MIN_ALLOCATED_ENGINES`	The minimum number of Engines that are allocated to this session at any time.
`OPERATION_TIMEOUT`	The maximum time an operation can take, in milliseconds.

SERVICE OPTION	DESCRIPTION
`TASK_FAIL_ON_OPERATION_FAILURE`	Determines whether an operation failure is considered to be a task failure by the Engine and Scheduler.
`NATIVE_NET_SERVICE`	Specifies whether to run the SpeedLink Service under .NET. Default is `true`. When set to `false` the SpeedLink Service runs under Java.

Standard Service Options in SpeedLink

The following table lists all current standard Service Options, and how they apply to a SpeedLink Service.

SERVICE OPTION	APPLIES TO SPEEDLINK
`PRIORITY`	Yes
`GRID_LIBRARY`	Yes
`KILL_ENGINE_ON_CANCEL`	Yes
`SERVICE_FAIL_RESTART`	Yes
`SERVICE_FAIL_RETRY`	No, retries must be performed by the client.
`ALLOWED_DRIVER_PROFILES`	Yes
`EMAIL`	Yes
`STATE_AFFINITY`	Yes
`AFFINITY_DEPTH`	No, there is no task discrimination.
`AFFINITY_WAIT`	Yes
`INVOCATION_MAX_TIME`	No, automatically calculated as `ALLOCATION_TIME` plus `OPERATION_TIMEOUT`. Prevents a runaway operation from taking an engine out.

SERVICE OPTION	APPLIES TO SPEEDLINK
`AUTO_CANCEL`	No
`MAX_TASK_RETRIES`	No, retries must be performed by the client.
`MAX_TASK_RESCHEDULES`	No, no redundant rescheduling.
`RESCHEDULE_ON_TIMEOUT`	No, no redundant rescheduling or retries.
`REDUNDANT_RESCHEDULING_ENABLED`	No, no redundant rescheduling.
`RESUBMIT_ON_DDT_FAILURE`	No, no DDT.
`COMPRESS_DATA`	No, the low-latency use-case does not include large data.
`ENCRYPTION_ENABLED`	No
`SHARED_UNIX_DIR`	No, no DDT.
`SHARED_WIN_DIR`	No, no DDT.
`MAX_ENGINES`	Yes, can be used to limit `MAX_ALLOCATED_ENGINES`. Overrides that if set.
`STATUS_EXPIRES`	Yes
`ENGINE_BLACKLISTING`	Yes
`FAILURES_BEFORE_BLACKLIST`	Yes
`CHECKPOINT`	No, no reason to checkpoint very short operations.
`PURGE_INVOCATION_DATA`	Yes
`AUTO_PACK`	No
`INVOCATIONS_PER_MESSAGE`	No
`JAR_FILE`	No

SERVICE OPTION	APPLIES TO SPEEDLINK
`SHARED_SERVICE`	No
`COLLECTION_TYPE`	No

Service-Oriented Integration Tutorial

- Introduction
  - Overview
  - The Integration Process
  - Argument Types
- A Simple Service in Java
  - The Service Implementation
  - Deployment and Grid Libraries
  - Registering a Service Type
  - The Client Application
  - Asynchronous and Parallel Processing Requests
  - Using a GridServer-Generated Proxy
- Container Bindings and Service State
  - About State
  - Container-managed Lifecycle
  - Cancel Method
  - Stateful Service Operations
  - Summary

Introduction

This tutorial explains the basic features of the GridServer Services-oriented integration approach by means of several sample programs. The code for the tutorial programs is available from the GridServer SDK, which can be downloaded from the **Download Components**panel in the top navigation bar of the Administration Tool.

Overview

GridServer hosts Services in an automatically scalable, load balanced, and fault-tolerant environment. Services can be written in a variety of languages and do not need to be compiled or linked with DataSynapse code.

Service implementations can be constructed using any of the following options:

- Arbitrary Java classes
- Arbitrary .NET classes
- Dynamic Libraries (.so, .dll) with methods that conform to a simple input-output string interface
- R functions
- Commands, such as a script or binary

These Services can be accessed in one of two ways:

- Service API — a client-side API in Java, COM, C++, R, or .NET
- Service Proxy — GridServer-generated C# or Java client stubs

The basic Service execution model is the same as that of other distributed programming solutions: method calls on the client are routed over the network, ultimately resulting in method calls on a remote machine, and return values make the reverse trip.

The chief benefit of hosting Services on GridServer over other approaches is that it virtualizes the Service. Rather than send a request directly to the remote machine hosting the Service, a client request is sent to the GridServer Manager, which enqueues it until an Engine is available. The first Engine to dequeue the request hosts the Service. Subsequent requests might be routed to the same Engine or might result in a second Engine running the Service concurrently. This decision is based on the Service's priority, the amount of resources the Service has received in comparison with other Services, and how much state related to the Service resides on the first Engine. If an Engine hosting a Service fails, another takes its place. This mechanism, in which a single virtual Service instance (the client-side object) is implemented by one or more physical instances (Engine processes) provides for fault tolerance and essentially unlimited scalability.

The Integration Process

Using the Service-Oriented integration approach involves six steps:

1. **Writing the Service, or adapting existing code**. A Service can be virtually any type of implementation: a library (DLL or .so), a .NET assembly, a Java class, an R function, even an

executable. No DataSynapse libraries need be linked, but the remotely callable methods of the Service might have to follow certain conventions. These conventions are described later.

2. **Deploying the Service**. The implementation files and other resources required by a Service must be accessible from all Engines. This can be accomplished with a shared file system or GridServer's file update mechanism.
3. **Registering the Service Type**. To make the Service Type visible to clients, it must be registered, using the Administration Tool.
4. **Creating a Service Session**. The client creates or gets access to a Service before using it — no discovery or binding is required. Each Service Session might have its own state. Because of virtualization, a single Service Session can correspond to more than one physical instance of the Service, such as more than one Engine running the Service's code. Multiple asynchronous calls to a Service usually result in more than one Engine creating and accessing those Services.
5. **Making requests**. The methods or functions of a Service can be called remotely either synchronously or asynchronously.
6. **Destroying the instance**. Clients must destroy a Service Session when they are done with it, to free resources.

Argument Types

Almost any public Java or .NET class with a public, no-argument constructor can be made into a Service with little or no modification. Each public method of the class can be accessed remotely through a Service operation. The input and output arguments must be either Serializable objects, support setter-getter interfaces in Java (beans) or public data members in .NET, or be pure strings — accessible as `String` in Java, `string` in .NET, `std::string` in C++, or `(char*, int)` in C. A dynamic library can be made into a Service if the functions exported as Service operations follow a simple string interface convention. The characteristics of each argument calling type are:

- **Serializable**— Services can use rich objects with serialization being accomplished through .NET and Java-specific data protocols. This is the simplest and most efficient way of exchanging data, but disallows interoperability between languages; for example, Java clients can only use Java Services and .NET clients can only use .NET Services.
- **XML Serializable**— .NET and Java objects can be written in such a way that they can be used with each other by converting them to XML.
- **Strings**— This approach allows for maximum interoperability among clients and Services implemented in different languages. For example, .NET clients can talk to Java or C++/C

Services, and so on. Also, automatic string conversion allows Java and .NET Services with non-string arguments to be called using String arguments.

A Simple Service in Java

This section describes how to create and access a simple Service in Java.

The Service Implementation

In this section, let us create a simple Service in Java. The implementation's class is called `JavaAdder` and has one method, `add`, that takes two `double` s. The method returns a `double` that is the sum of its arguments.

```
package examples.adder.service.JavaAdder;

public class JavaAdder {

    public double add(double a, double b) {

        return a + b;

    }

}
```

Deployment and Grid Libraries

Once the class is written and compiled, it must be deployed as a Grid Library.

Grid Libraries are the method of deploying resources to Engines. They are an archive containing a set of resources and properties necessary to run a Grid Service, along with configuration information that describes how those resources are to be used.

To create and deploy a Grid Library:

1. Create a JAR file containing the necessary class file (`JavaAdder.class`).
2. Create a `grid-library.xml` file, like the one below:

```
<?xml version="1.0" encoding="UTF-8"?>

<grid-library>
```

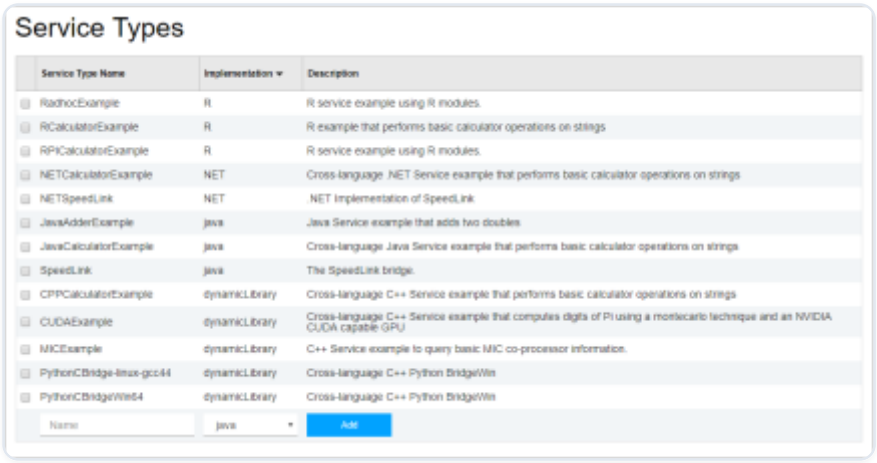
```
    <grid-library-name>adder</grid-library-name>
    <grid-library-version>1.0.0.1</grid-library-version>
    <jar-path>
        <pathelement>jars</pathelement>
    </jar-path>
</grid-library>
```

3. Create an `adder` directory in a temporary location, and place the `grid-library.xml` file in it.
4. Create a `jars` directory within the `adder` directory, and place the JAR file containing the class in it.
5. Zip all contents under `adder` directory, making sure the directory `adder` is not included in the ZIP archive, to create an `adder-1.0.0.1.zip` Grid Library. If you're using UNIX, you can also create a `tar.gz` file of the directory instead (such as `adder-1.0.0.1.tar.gz` or `adder-1.0.0.1.tgz`).
6. Log in to the GridServer Administration Tool and go to **Services > Services > Grid Libraries**.
7. At the top right corner of the page, click **Upload Grid Library**, browse to and select your Grid Library file, then click **Upload**.
8. Select your Grid Library in the list and click **Deploy**.

More information about Grid Libraries and the deployment process is described the TIBCO GridServer® Administration and the TIBCO GridServer® Developer's Guide.

Registering a Service Type

Service Types must be registered from the Administration Tool, at **Services > Services > Service Types**.



A list of existing Service Types appears on that page, along with a line for adding a new Service Type. Enter the Service Type name on the blank line. Let us name the Service Type `JavaAdderExample`. Now select Java as the Service implementation, then click **Add**.

In the window that appears after clicking the **Add**button, enter the fully qualified class name for the Service Type, which in this case is `examples.adder.service.JavaAdder`. The window also allows you to enter options for the Service Type. Enter a `*` in the **serviceMethods**field, indicating that all public methods might be called as Service Type methods.

> **Note:**
>
> You must set the Grid Library's name in the Service Type's **gridLibrary**field.

The Client Application

Having deployed and registered the Service Type, we are now ready to use it. There are two different techniques we can use to access this Service from a client:

- The Service API
- GridServer-generated proxy

Let us first demonstrate accessing the Service using the Service API in Java. Refer to the Javadocs on the **Documentation**panel in the top navigation bar of the Administration Tool. The `Service` class is used to access the Service either synchronously or asynchronously. This section consists largely of small code snippets. All of the code can be found in a single class, `example.adder.client.AdderClient`.

Only a few lines of code are needed to use the `JavaAdder` Service:

```
// Create a Service instance of JavaAdderExample

Service s =
ServiceFactory.getInstance().createService("JavaAdderExample");

// Perform Synchronous add

Object[] arguments = new Object[] { new Double(5), new Double(6) };

Double sum = (Double)s.execute("add", arguments);

System.out.println("Result of add: " + sum);
```

The first line gets an instance of the `ServiceFactory` class and calls its `createService` method to create a `Service` instance for the Service. If you try to create a Service Type whose name was not registered, `createService` throws a `GridServerException`.

The second line prepares the arguments to be submitted to the Service, two `Double` s (note the use of the primitive wrapper object). The third line executes the `add` method with the given argument list and blocks until the method completes and its return value makes its way back to the client. If the remote method throws an exception, or an error occurs in the course of processing the request, an exception is thrown. `GridServerException` might wrap another exception; use its `getCause` method to obtain the wrapped exception.

A Java client does not stop automatically, and pends after submitting Services and collecting results. Instead of calling `System.exit()` to stop the client, you can set a property to change this behavior. Add this to your code:

```
DriverManager.setProperty(DriverManager.IS_DAEMON,

                          Boolean.TRUE.toString());
```

When this property is set to true, all client threads are daemon threads, meaning that the allow the process to shut down when all threads have shut down.

Asynchronous and Parallel Processing Requests

A client can use the `submit` method instead of the `execute` method of `Service` to perform a remote invocation without waiting for the result. This allows the caller to make many asynchronous requests in parallel — with these requests being executed in parallel if the resources are available

to do so. In addition to the method name and argument, the `submit` method takes a callback object that implements the `ServiceInvocationHandler` interface. This interface has a `handleResponse` method that is called with the method's return value in the event of a normal response, and a `handleError` message that is called if an error occurs. The submit method returns an integer that uniquely identifies the particular call; this unique ID is passed as the second argument to the `ServiceInvocationHandler` methods, so that you can match the response with the request if need be.

To use the submit method, we first require a class implementing `ServiceInvocationHandler`. The `handleResponse` method displays the response and adds it to a total.

```
// Handler for Service Clients

static class AdderHandler implements ServiceInvocationHandler {

    public void handleError(ServiceInvocationException e, int id) {

        System.out.println("Error from " + id + ": " + e);

    }

    public void handleResponse(Serializable response, int id) {

        System.out.println("Response from " + id + ": " + response);

        _total += ((Double)response).doubleValue();

    }

    public double getTotal() {

        return _total;

    }

    private double _total = 0;

}
```

Now we can invoke the `submit` method:

```
// Perform Asynchronous adds

AdderHandler handler = new AdderHandler();

for (int i=0; i < 10; i++) {

    s.submit("add", new Object[] { new Double(i), new Double(i) },
handler);

}

// Wait until all the invocations have returned

s.waitUntilInactive(0);

System.out.println("Total: " + handler.getTotal());
```

This code first creates a `ServiceInvocationHandler`, and then it calls `submit` several times. To wait for all the responses to arrive, the call to `waitUntilInactive` causes the current thread to wait until all outstanding requests have finished. The argument is a timeout value in milliseconds; an argument of zero means wait indefinitely.

Using a GridServer-Generated Proxy

Using the Service API is easy and flexible, but there is an easier way to access Service implementations you write and deploy on GridServer. After registering the Service Type using the Administration Tool, a menu action becomes available on the **Services > Services > Service Types**page for creating client-side access proxies in either Java or C#.

There are several advantages to using proxies:

- **No manual coding required**— the proxy code itself does the argument passing and return value casting. It's a lot easier to use a class that looks like your Service!
- **Type-safe**— Argument and return types are checked at compile-time, instead of on the Engine at runtime.
- **Vendor Neutral**— Application code does not need to have compile-time dependency with any DataSynapse-specific libraries if the proxies are used. The proxies themselves make use of DataSynapse libraries, but do not expose this dependency to your application.

The following code demonstrates calling the `add` method on the proxy object:

```
// Create an instance of the Service Proxy

JavaAdderProxy adder = new JavaAdderProxy();

// Perform Synchronous add

double sum = adder.add(8.0, 10.0);

System.out.println("Result of Add: " + sum);
```

So what's going on behind the scenes? Here is a look at the `JavaAdderProxy` code for the method `add`:

```
public double add(double in0, double in1) throws Exception {

    return ((java.lang.Double) execute("add", new Object[] {new
java.lang.Double(in0), new java.lang.Double(in1)})).doubleValue();

}
```

If you want to call this method asynchronously (possibly to have many requests done in parallel) you can create a callback class that implements an interface found in the proxy. The callback interface defined in the proxy is found below:

```
public interface Callback extends ServiceBindingStub.AsyncCallback {

    public void handleResponse(Object response, int id);

    public void handleError(Exception e, int id);

}
```

Use this class in a fashion similar to that of example 3.3. Instead of implementing the `ServiceInvocationHandler` class you must implement the `JavaAdderProxy.Callback` class. The benefit to this approach, however, is that the code you write for this callback does not have to import any DataSynapse classes. All you need to do is import your proxy and use it.

When using a GridServer-generated proxy, asynchronous calls are performed by invoking an overloaded method that has the same name and arguments as the synchronous version, but adds an additional argument for the callback. The proxy also has a `waitUntilInactive` method to

pause the current thread until all outstanding asynchronous calls have completed. The following example illustrates this:

```
// Perform Asynchronous adds

AdderCallback callback = new AdderCallback();

for (int i=0; i < 10; i++) {

    adder.add(i, i, callback);

}

// Wait until all the invocations have returned

adder.waitUntilInactive();

System.out.println("Total: " + callback.getTotal());
```

Container Bindings and Service State

In this section we discuss the relationship between the container (GridServer Engine), the Service implementation (class or library), and operations that are exposed by the Service.

About State

The GridServer technology allows Services to have a state—per-client or global—that is preserved by the container and is consistent across all physical Service instances, that is, all Engine-side instantiations of a client-side Service instance. The latter concept is important to understand: since many GridServer Engines can perform Service requests in parallel, the Service deployer must inform GridServer which methods are responsible for state and which are not. This way, GridServer can ensure that the object’s state is up-to-date prior to servicing the next request. In addition to the state, the container must know which methods to expose as Service operations and which methods to call for initialization and destruction of the Service implementations in memory.

Container-managed Lifecycle

Services hosted in the GridServer environment are virtualized—provisioned dynamically on Engines at the behest of the GridServer Manager to satisfy demand. In this model, Service initialization and destruction happens automatically, but proper start-up and clean-up operations can be performed on the Service by registering the right methods to call on the Service to

accomplish these tasks. The following simple class demonstrates a Service that connects to a hypothetical database:

```
public class DBCalculator {

    public void initDBConnection(Properties connProps, int someVal) {

        _connection = new Connection(Properties props);

    }

    public Example calculate(String arg1, …) { … }

    public void close() {

        if (_connection != null) _connection.close();

    }

    private Connection _connection;

}
```

When a class is registered as a Service Type, it is possible to specify in the **ContainerBinding**section of the registry page which methods are used for initialization and destruction of the Service. In this example, `initDBConnection` is specified as the “initMethod” field and `close` is specified in the “destroyMethod” field. Since `initDBConnection` takes two arguments, a Properties object and an `int`, the client-side proxy class has a constructor with the same argument list:

```
public class DBCalculatorProxy {

    public DBCalculatorProxy(Properties connProps, int someVal) { … }

    …

}
```

The “close” method is listed as the `destroyMethod`, and is exposed on the client-side proxy as a method named “destroy.”

Cancel Method

If a Service is canceled by the user or a Service request (task) is canceled, the managing container receives a message and calls a specified operation on the Service automatically. For a given Service, a method can be assigned to be called by the container under this cancellation event. Here's a simple example:

```
public class DBCalculator {

    …

    public void stop() {

        _isStopped = true;

    }

    public Bar calculate(String arg1, …) {

        while (_isStopped != null) {

           // iterate over looping variable

        }

    }

    private volatile boolean _isStopped = false;

}
```

In this example, the `stop` method must be registered as the "cancelMethod". If it is not possible to stop the operation in this way, GridServer allows canceled Service operations to cause the entire Engine process to be killed and restarted automatically. The Service option is called `KILL_ENGINE_ON_CANCEL` and takes the value true or false.

Stateful Service Operations

Now let us demonstrate how to manage state in the Service instances and how to instruct GridServer to behave properly under these operating conditions. First, let's consider the following bond calculator class that holds state:

```
public class BondValuationService {

    public void addBonds(String[] bondIds) {…}

    public void setBonds (String[] bondIds) {…}

    public ValuationResults valueAllBonds(Scenario s) {…}

}
```

Both the `addBonds` and `setBonds` methods take a list of bond IDs. We assume that in these methods the actual object representations are constructed and the bond data is fetched from an appropriate source and loaded into the object representations. We further assume that there is a fair amount of latency and computation involved in these object constructions—this is why the writer of this Service is interested in keeping this stateful data present in the Service. There is also a stateless computing method that computes the valuations for these financial instruments under different market conditions or scenarios.

In this example, we see that there are two stateful methods— `addBonds` and `setBonds`. But their behavior is different: if `addBonds` is called, the new bonds are added to the existing collection, while the `setBonds` method replaces the existing collection. One can capture this distinction and ensure proper behavior of the GridServer deployment by setting the appropriate fields in the **ContainerBinding**section of the **Service Types**page. There is a field called `appendStateMethods` that can be set to the value `addBonds` and a field called `setStateMethods` that can be set to the value `setBonds`. More than one method can be listed, separating them by commas.

The client-side proxy class generated by GridServer looks quite similar to the actual Service:

```
public class BondValuationServiceProxy {

    public void addBonds(String[] bondIds) {…}

    public void setBonds (String[] bondIds) {…}

    public ValuationResults valueAllBonds(Scenario s) {…}

}
```

However, if you are using the Service API, you have to make the distinction in the API calls themselves. Notice the following use of the method `updateState`:

```
Service service =
ServiceFactory.getInstance().createService("bondCalculator");

service.updateState("addBonds", new Object[] { bondIds }, false);

// or

service.updateState("setBonds", new Object[] { bondIds }, true);
```

The last argument to the `updateState` method is a Boolean that indicates whether this state must replace the current state (true) or append the current state (false).

Summary

- You can initialize and destroy Service state by writing methods in your Service that perform the required actions, and setting the `initMethod` and `destroyMethod` options to the names of the methods.
- The client-side proxy creates a constructor with the same method argument types as the `initMethod` that was specified in the Service Type Registry.
- The client-side proxy contains a destroy method that calls the method specified as the `destroyMethod` in the Service Type.
- The Service deployer must also specify which Service methods change state, and whether they append or set the state. These methods are called normally from a proxy, and via `updateState` using the Service API.