TIBCO DataSynapse GridServer® Manager

Introducing TIBCO GridServer®

Version 7.1.0
July 2022

Document Updated: September 2022

Copyright © 2001-2022. TIBCO Software Inc. All Rights Reserved.

Contents

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

Grid Client Interfaces
C++/C
Batch Clients
R functions

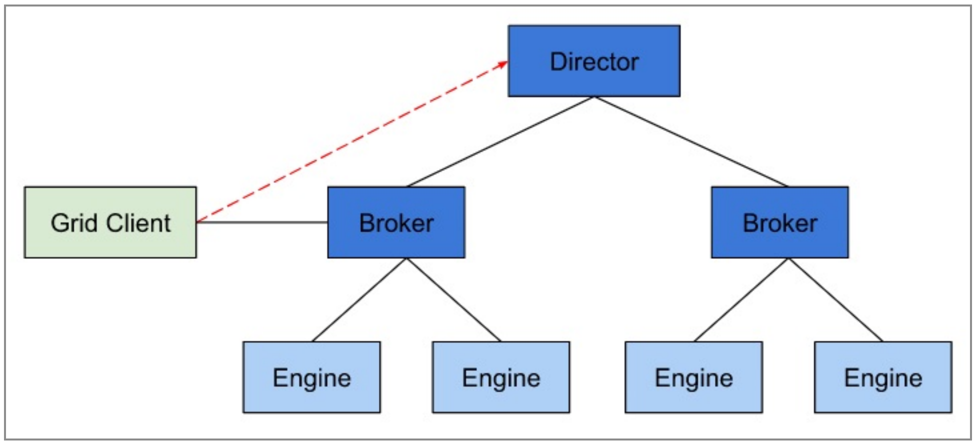
GridServer enables a truly Service-oriented approach to Grid computing. The Grid client creates a Service instance and calls operations (methods) on the Service. These calls are virtualized and invoked on multiple Grid nodes. This virtualization provides limitless scalability for hosting these Services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

Services are described in more detail in GridServer Services.

GridServer Architecture

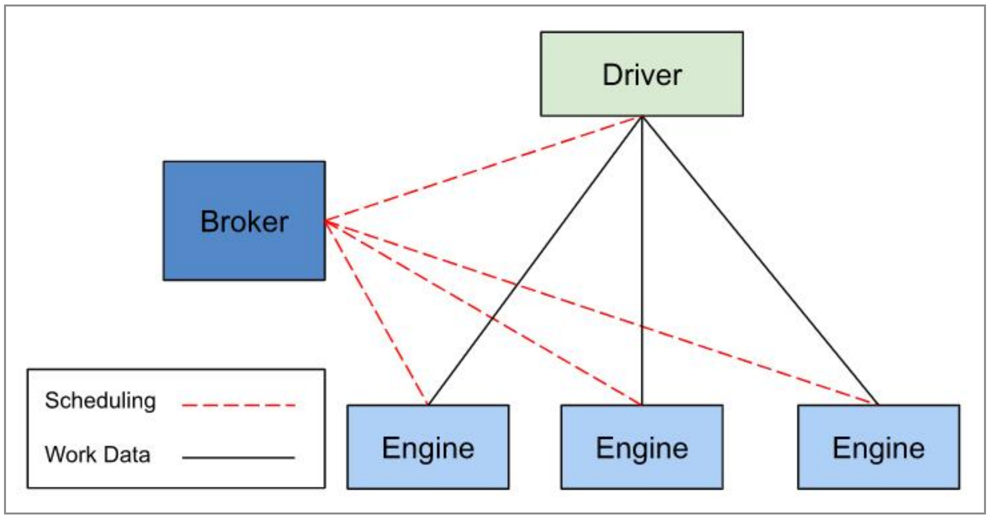
DataSynapse GridServer has four different components in its architecture:

- **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.

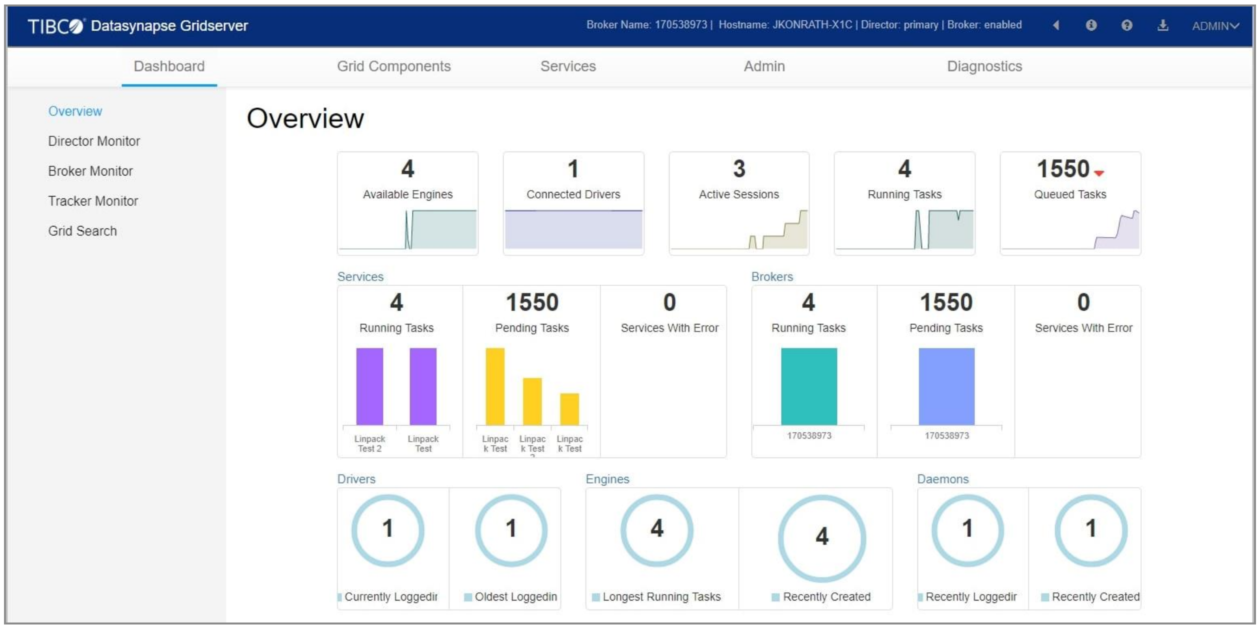
Grid Clients submit requests through the Broker. Engines receive work requests from the Broker, and in most cases, exchange data directly with the Grid Clients. This allows the system to be highly scalable. Since the Brokers manage all work requests, load balancing is optimal and resilience is built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

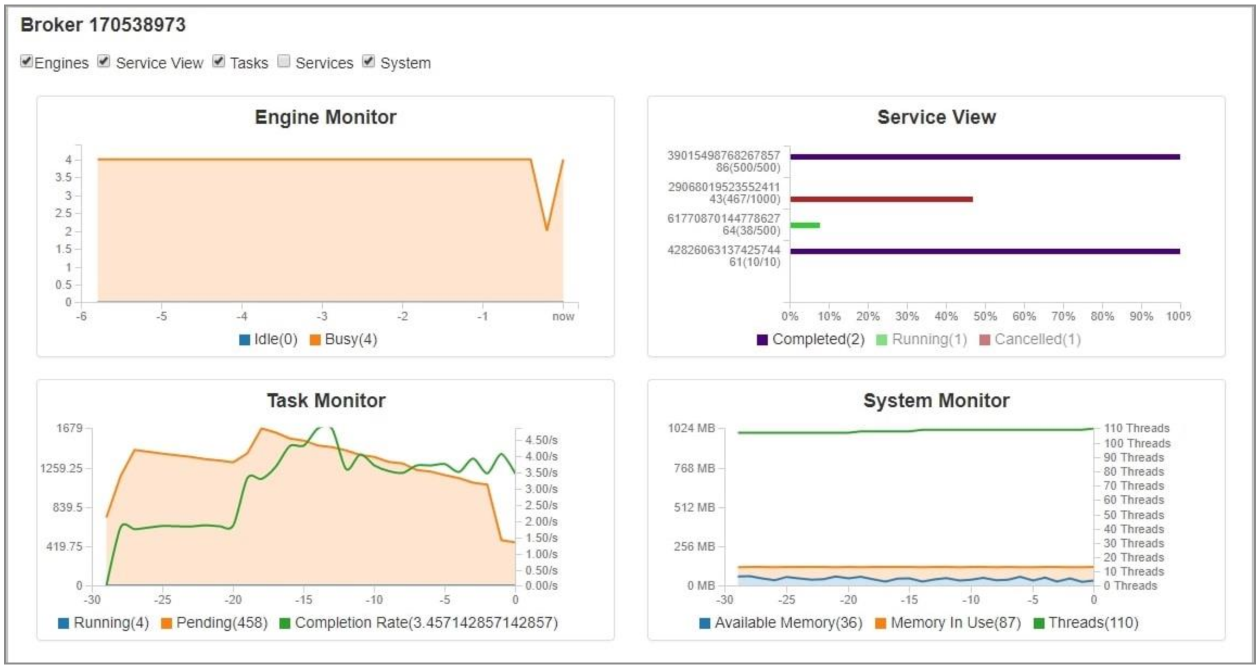
GridServer includes a highly manageable operating environment featuring a web-based console, the GridServer Administration Tool, and a SOAP-based administration interface. Both allow you to deploy Services, manage the workloads running on the Grid, and configure the GridServer environment. The screenshot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



Broker 170538973

✓ Engines ✓ Service View ✓ Tasks ☐ Services ✓ System

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.

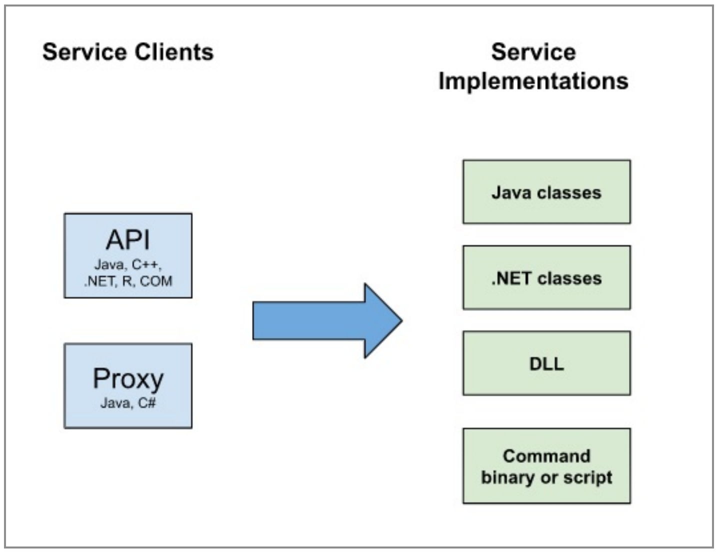
The Service-Oriented method uses two components: Clients and Service Implementations. Descriptions of both follow.

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.
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.

Setting	Description
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.

Warning 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 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.

Warning 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 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:

Procedure

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 Page** box 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 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 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 Configurations**, 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 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 check box.
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 > 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 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 further 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 > Engines > 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.

Inside the GridServer Director

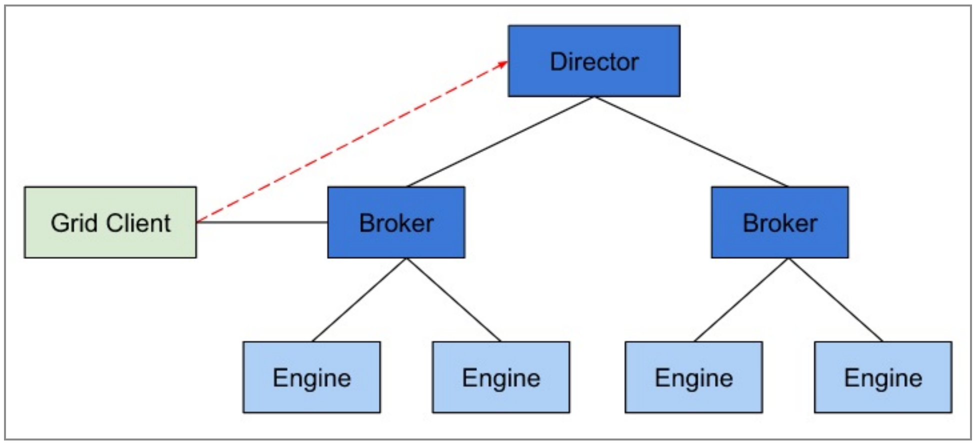
The GridServer Director is part of a GridServer Manager, the controlling mechanism of the Grid. Directors route Engines and Drivers to Brokers, and manage Bro-kers and Engine Daemons. Directors balance the load among their Brokers and provide Broker fault tolerance.

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.

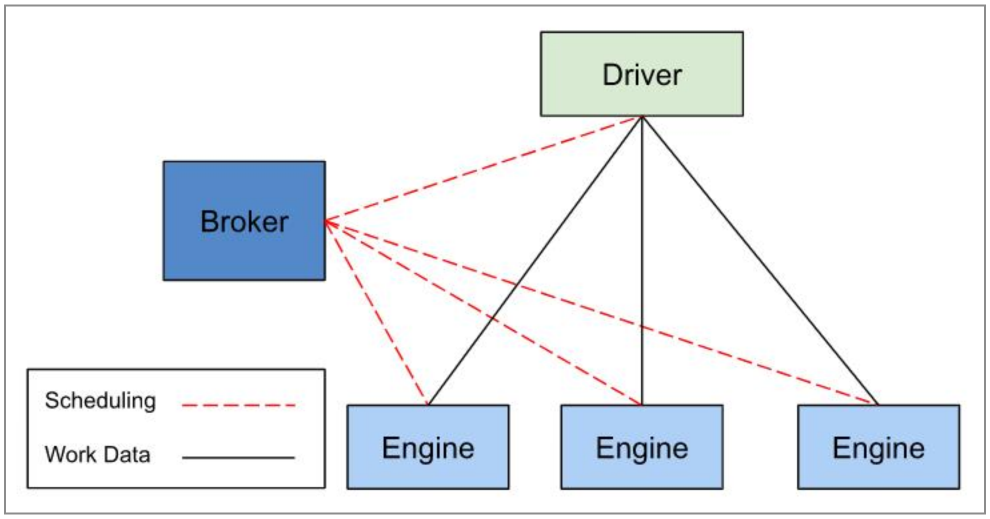
Resources are distributed with *Grid Libraries*. A Grid Library is an archive containing a set of resources and properties necessary to run a Grid Service, along with configuration information that describes how those resources are to be used. Grid Libraries can contain JARs, native libraries, configuration 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.

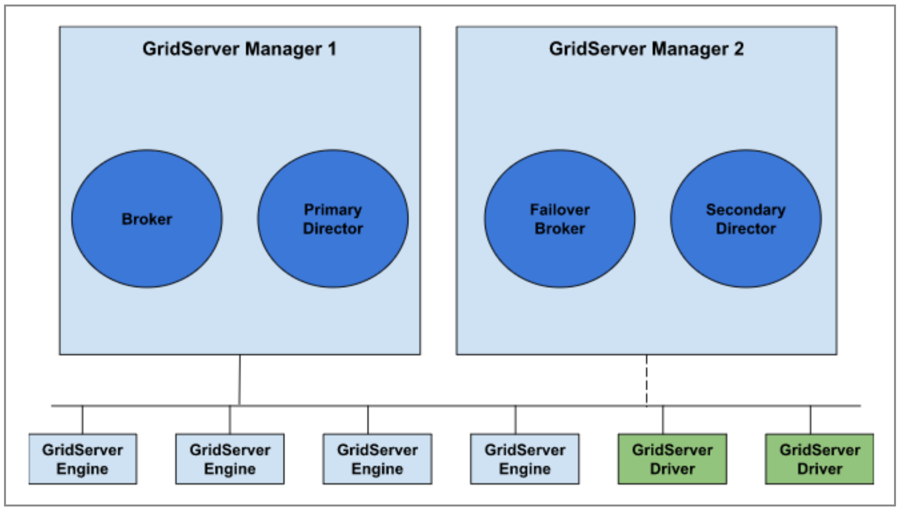
Inside GridServer Topology

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.



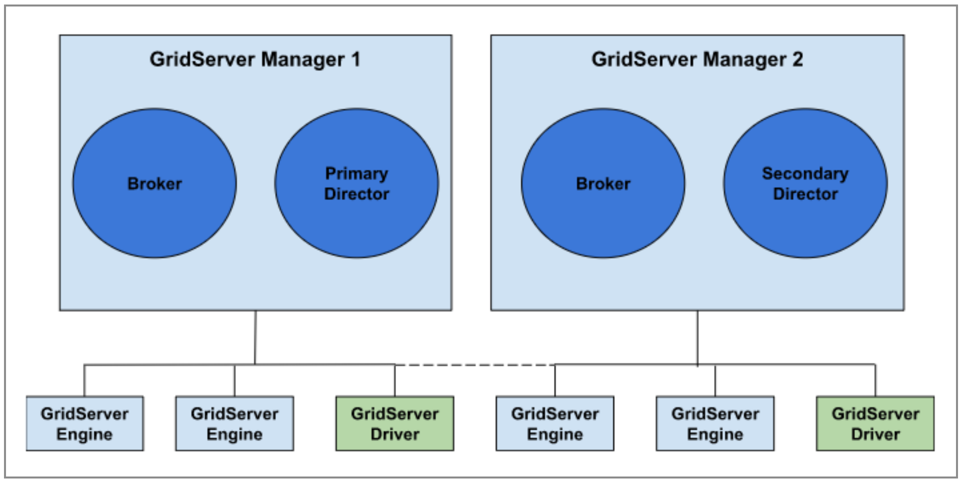
To prevent this, redundancy can be built into the GridServer architecture. One method is to run a second Manager with a Secondary Director, and configure Engines and Drivers with the address of both Directors. If the Primary Director fails, Engines and Drivers contact the Secondary Director, which contains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

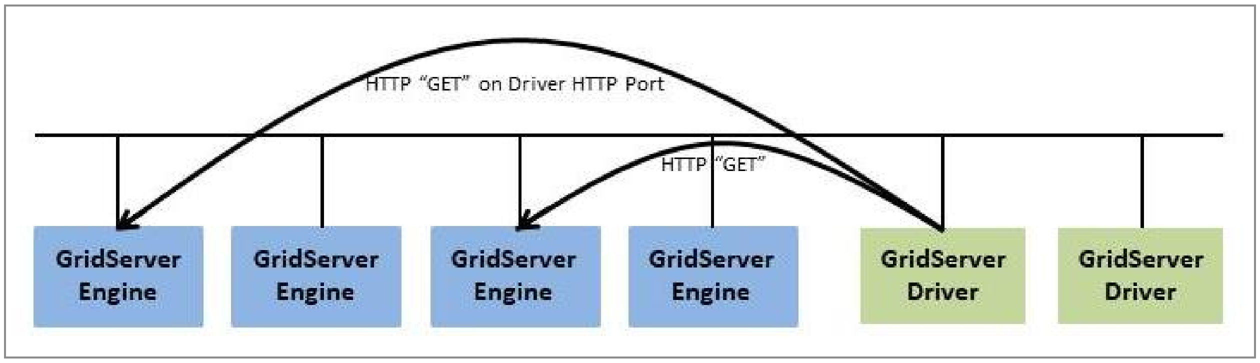
- Instead of using one Grid for all types of Services, you might wish to divide different subsets of Services (for example, by size or priority) to different Brokers.
- Your network might dictate how your Manager environment must be planned. For example, if you have offices in two parts of the country 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.

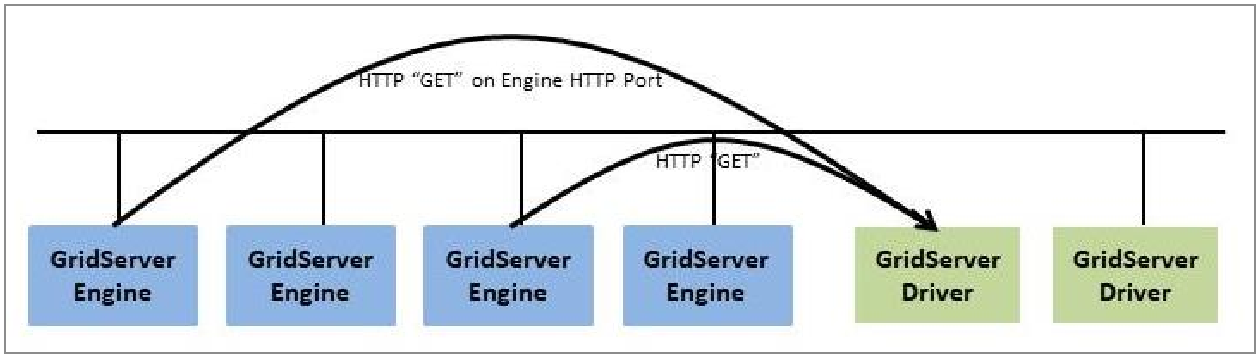
With this flexibility, it’s possible to architect a Manager model to provide a work space that facilitates your workload and traffic. For more information about how to best design your Manager environment, contact our Integration Services staff, and we can help you determine how to best configure your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

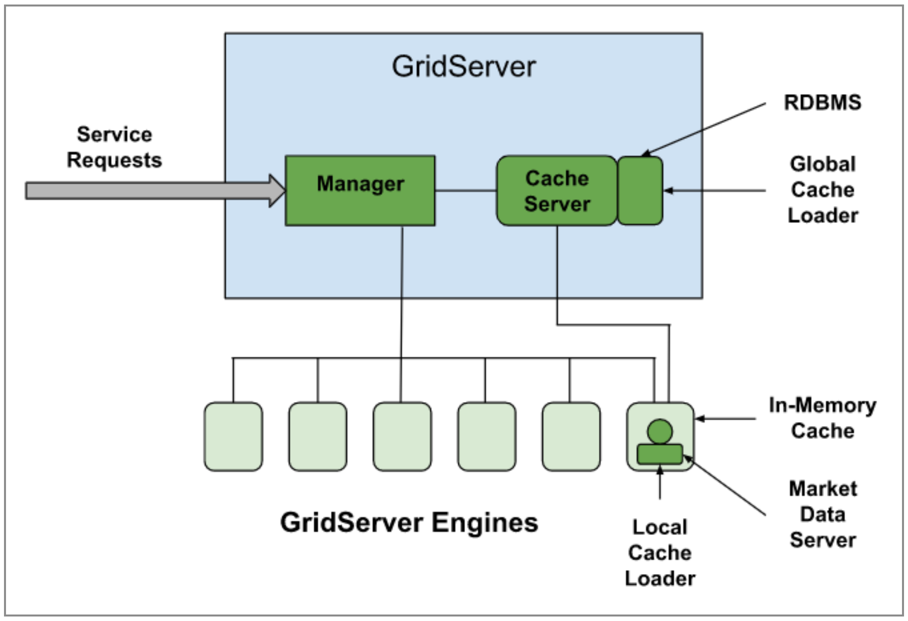


GridCache

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 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.

Client Interface	Description
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

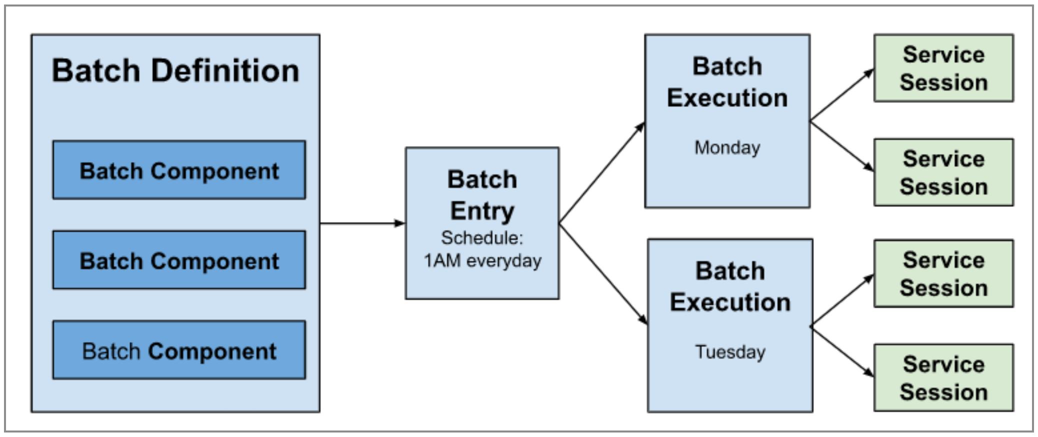
GridServer comes with a built-in batch scheduling facility that makes it easy to schedule Grid workflows from a Web browser. Services can be scheduled from this facility, as well as Grid administrative functions. For example, a user could schedule all the Engines to reconfigure each night at a specific 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

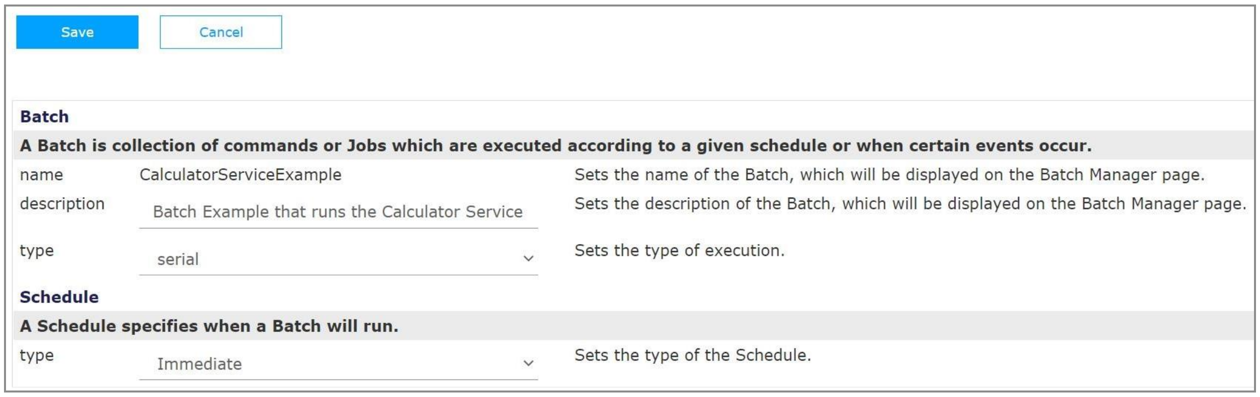
A *Batch Definition* contains instructions in the form of components that define both the schedule and the components that are executed. When the Batch Definition is scheduled on the Manager, it creates a *Batch Entry*, which typically waits until its scheduled time, then creates a *Batch Execution*. 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.



Batch

A Batch is collection of commands or Jobs which are executed according to a given schedule or when certain events occur.

name	CalculatorServiceExample	Sets the name of the Batch, which will be displayed on the Batch Manager page.
description	Batch Example that runs the Calculator Service	Sets the description of the Batch, which will be displayed on the Batch Manager page.
type	serial	Sets the type of execution.

Schedule

A Schedule specifies when a Batch will run.

type	Immediate	Sets the type of the Schedule.

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*.

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

Documentation for TIBCO GridServer® is available on the TIBCO GridServer® Product Documentation page.

The following documents for this product can be found in the TIBCO Documentation site:

- TIBCO GridServer® Release Notes
- TIBCO GridServer® Installation
- TIBCO GridServer® Introducing TIBCO GridServer®
- TIBCO GridServer® Administration
- TIBCO GridServer® Developer's Guide
- TIBCO GridServer® Upgrade
- TIBCO GridServer® Security
- TIBCO GridServer® COM Integration Tutorial
- TIBCO GridServer® PDriver Tutorial
- TIBCO GridServer® Speedlink
- TIBCO GridServer® Service-Oriented Integration Tutorial

How to Contact TIBCO Support

Get an overview of TIBCO Support. You can contact TIBCO Support in the following ways:

- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support website.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to TIBCO Support website. If you do not have a user name, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, GridServer, FabricServer, GridClient, FabricBroker, LiveCluster, and SpeedLink are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (https://www.tibco.com/patents) for details.

Copyright © 2001-2022. TIBCO Software Inc. All Rights Reserved.