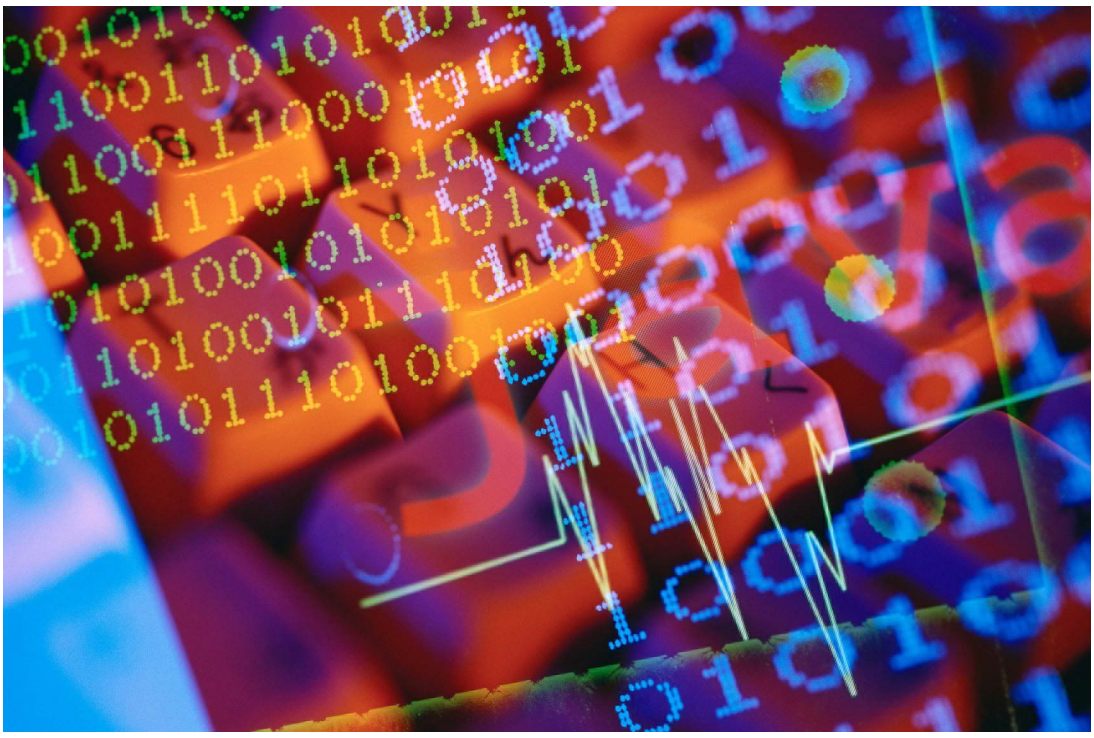
GridServer Developer's Guide

Version 4.2



The GridServer Developer's Series

Proprietary and Confidential

Confidentiality and Disclaimer

Neither this document nor any of its contents may be used or disclosed without the express written consent of DataSynapse. This document does not carry any right of publication or disclosure to any other party.

While the information provided herein is believed to be accurate and reliable, DataSynapse makes no representations or warranties, express or implied, as to the accuracy or completeness of such information. Only those representations and warranties contained in a definitive license agreement shall have any legal effect. In furnishing this document, DataSynapse reserves the right to amend or replace it at any time and undertakes no obligation to provide the recipient with access to any additional information. Nothing contained within this document is or should be relied upon as a promise or representation as to the future.

This product includes software developed by the Apache Software Foundation (www.apache.org/).

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (www.openssl.org/).

This product includes code licensed from RSA Data Security (java.sun.com/products/jsse/LICENSE.html).

DataSynapse GridServer Developer's Guide Version 4.2

DataSynapse, Inc. 632 Broadway, 5th Floor; New York, NY 10012
Tel: 212.842.8842 Fax: 212.842.8843
Email: info@datasynapse.com Web: www.datasynapse.com

For technical support issues and product updates, please visit customer.datasynapse.com.

We appreciate any comments or suggestions you may have about this manual or other DataSynapse documentation. Please send your feedback to docs@datasynapse.com.

04220102105

Contents

Chapter 1
Introduction

This guide is your reference for developing applications that utilize GridServer installations. It is divided into several sections to help you understand the principles of the GridServer system, and how to program using the GridServer API.

Before you begin

This guide assumes that you already have a GridServer Manager running and know the hostname, username, and password. If this isn't true, see the *GridServer Installation Guide* or contact your administrator.

GridServer 4.2 Documentation Roadmap

The following documentation is available for GridServer 4.2:

GridServer Guides

Four guides and four tutorials are included with GridServer in Adobe Acrobat (PDF) format. They are also available in print format. To view the guides, log in to the Administration tool, select the **Admin** tab, go to the **Documentation** page, and select a guide. A search engine is also available on this page for you to search all of the documentation for a phrase or keywords. The PDF files can also be found on the Manager at `livecluster/admin/docs`. The following guides are available:

Introducing the GridServer Platform Series:

Introducing the GridServer Platform	Contains an introduction to GridServer, including definitions of key concepts and terms, such as work, Engines, Directors, and Brokers. This should be read first if you are new to GridServer.

The GridServer Administration Series:

GridServer Administration Guide	Covers the operation of a GridServer installation as relevant to a system administrator. It includes basic theory on scheduling, fault-tolerance, failover, and other concepts, plus howto information, and performance and tuning information.
GridServer Installation Guide	Covers installation of GridServer for Windows and Unix, including Managers, Engines, and pre-installation planning.

The GridServer Developer Series:

GridServer Developer's Guide	Contains information on how to develop applications for GridServer, including information on using Services, PDriver (the Batch-oriented GridServer Client), the theory behind development with the GridServer Tasklet API and concepts needed to write and adapt applications.
GridServer Object-Oriented Integration Tutorial	Tutorial on developing applications for GridServer using the object-oriented Tasklet API in Java or C++.
GridServer Service-Oriented Integration Tutorial	Tutorial on developing applications for GridServer using Services, such as Java, .NET, native, or binary executable Services.
GridServer PDriver Tutorial	Tutorial on using PDriver, the Parametric Service Driver, to create and run Services with GridServer.
GridServer COM Tutorial	Tutorial explaining how client applications in Windows can use COMDriver, GridServer's COM API, to work with services on GridServer.

Other Documentation and Help

In addition to the GridServer guides, you can also find help and information from the following sources:

GridServer Administration Tool Help Context-sensitive help is available throughout the GridServer Administration Tool by clicking the help icon located on any page. This provides reference help, plus how-to topics.

API Reference Reference information for the GridServer API is provided in the GridServer SDK in the `docs` directory. The Java API information is in JavaDoc format, while C++ documentation is presented in HTML, and .NET API help is in HTMLHelp. You can also view and search them from the GridServer Administration Tool; log in to the Administration Tool, click the **Admin** tab, and select the **Documentation** link.

Knowledge Base A searchable archive of known issues and support articles is available online. To access the DataSynapse Knowledge Base, go to the DataSynapse customer extranet site at `customer.datasynapse.com` and log in. You can also use this site to file an issue report, download product updates and licenses, and view documentation.

Document Conventions

Convention	Explanation	Example
italics	Book titles	The *GridServer Developer's Guide* describes this API in detail.
"Text in quotation marks"	References to chapter or section titles	See "Preliminaries."
bold text	Emphasizes key terminology	Client applications (**Drivers**) submit work to a central **Manager**.
	Interface labels or options	Enter your URL in the **Address** box and click **Next**.
`Courier New`	User input, directories, file names, file contents, and program scripts	Run the script in the `/opt/datasynapse` directory.
Blue text	Hypertext link. Click to jump to the specified page or document.	See the *GridServer Developer's Guide* for details.
`[GS Manager Root]`	The directory where GridServer is installed, such as `c:\datasynapse` or `/opt/datasynapse`.	The Driver packages are located in `[GS Manager Root]/webapps/livecluster/WEB-INF/driverInstall`

Chapter 2
GridServer Application Development

This section of the *GridServer Developer's Guide* is your starting point for developing applications that utilize your GridServer installation. The document is divided into several chapters to help you understand the principles of the GridServer system, and how to program applications utilizing GridServer.

GridServer Programming Options

There are several options available to you when you adapt your applications to use GridServer. The following sections describe how to use each of them.

Services

Services provide for remote execution of code in a way that is scalable, fault-tolerant, dynamic and language-independent. Services can be written in a variety of languages and do not need to be compiled or linked with DataSynapse code. There are client-side APIs to create Service Sessions using Java, C++, COM, and .NET, as well as a Web Services interface. A Service object on a client can create and use a Service implemented in the same or another languages. In the Service model, requests on the client are routed over the network, ultimately resulting in invocations on a remote machine, and response values make the reverse trip.

With GridServer, Services are **virtualized**; rather than send a request directly to the remote machine hosting the Service Session, a client request is sent to the GridServer Manager, which enqueues it until an appropriate Engine is available. The GridServer Manager selects which Engine should service a request. The first Engine to dequeue the request hosts the Service Session. Subsequent requests may be routed to the same Engine or may result in a second Engine running the Service concurrently. (For information on how this decision is made see Chapter 5, "Scheduling" on page 29 of the *GridServer Administration Guide* for details.) If an Engine hosting a Service Session should become unavailable, another will take its place. This mechanism, in which a single *virtual* Service Session is implemented by one or more *physical* Sessions (Engine processes) provides for fault tolerance and essentially unlimited scalability.

Chapter 3, "Creating Services" on page 23 details how to implement Services; Chapter 4, "Accessing Services" on page 33 explains how to utilize Services in your application.

The Tasklet API

The Tasklet API, available in Java, using JDriver, and C++, using the CPPDriver, is a forerunner to the Services approach, and is suitable when your application code on both Driver and Engine are written in C++ or Java, the work you will distribute can be logically broken down into units of work that can run independently and combine for a final result, and your application can be refactored to include GridServer API calls directly in your code.

The Tasklet API consists of four types of objects: `Tasklet`, `TaskInput`, `TaskOutput`, and `Job`.

A `Tasklet` is the Service implementation that is created on the Engine side. It packages the computation's common data and behavior needed to run one unit of work in the overall problem being distributed. A `Tasklet` is a concrete Service object that contains a method for doing the work, as well as other methods for lifecycle management. A `Tasklet` takes a `TaskInput` as input, operates on it, and produces a `TaskOutput` as output. A `TaskInput` packages the data and code that is unique to one work unit in the overall computation, and the `TaskOutput` packages the results of an individual unit of work.

A `Job` object, represents the overall group of work being computed. The Job is the coordinator of the individual work units or tasks. Using a Job object, your application creates a Job specific Tasklet, submits `TaskInput`s, and processes the `TaskOutput`s as they arrive.

Chapter 5, "The Tasklet API" on page 45 explains how to use the Tasklet API.

PDriver

The Parametric Job Driver, or PDriver, is a Driver that can execute command-line programs as a parallel processing service using the GridServer environment. This enables you to write a simple script to run a program on several Engines, and return the results to a central location.

PDriver scripts, which are written in the PDS scripting language, enable you to run the same program on Engines several times with different parameters. A script is used to define how these parameters change.

One way PDriver scripts can achieve parallelism is to iteratively change the value of variables that are passed to successive tasks as parameters. A script can step through a range of numbers and use each value as a parameter for each task that is created. Or, a variable can be defined containing a list of parameters.

Chapter 6, "PDriver" on page 49 explains how to use PDriver.

Resource Deployment

Service Deployment files that are used by Engines are centrally managed, starting at the Director. Files can be uploaded to the Director via the **Resource Deployment** page on the **Services** tab in the Administration Tool. The resources centrally located on Director are then synchronized to Brokers, which then synchronize them with Engines.

Grid Libraries (or **GLs**) are the enterprise-level method of deploying resources to Engines. They are an archive containing a set of resources and properties necessary to run a Grid Service, along with configuration information that describes how those resources are to be used. Grid Libraries can contain Java classes and JARs, native libraries, .NET assemblies, configuration files, Java system properties, Engine hooks, and alternate JREs needed to run a Service. They can also contain references to other GLs as dependencies. A Service Session can use a GL by setting the appropriate options for the Service Type used by the session.

The `tools/grid-library` directory of the SDK includes an example ANT build script that can be used to build Grid Libraries. The `services` examples in the SDK can be automatically packaged as Grid Libraries by using this script and included configuration files. Each service example contains `grid-library.xml` and `grid-library-build-properties` files. The tools directory contains `build.xml`, `build.bat` and `build.sh`, which parse the `grid-library-build-properties` files to create Grid Libraries.

For more information on packaging and deploying Grid Libraries, see Chapter 7, "Application Resource Deployment" on page 43 in the *GridServer Administration Guide*.

Logging and Debugging

GridServer contains comprehensive logging facilities on Engines. This can be used to diagnose problems with Services running on Engines, and your application can write information to these logs. This section contains an overview of GridServer's log facility, plus information on using it from your application, and how to attach a debugger to an Engine, if needed.

Log Overview

The DataSynapse logger is used to provide diagnostics messages to the console and to file. This section covers how to access these logs, and how to interface with the loggers.

The DataSynapse logger is based on the `java.util.logging.Logger` model, in terms of its log levels. A quick overview of levels, in order:

Level	Description
Severe	Indicates serious failures
Warning	Indicates potential problems
Info	Displays informational messages
Config	Displays static configuration messages
Fine	Provides tracing information
Finer	Indicates a fairly detailed tracing message
Finest	Indicates a highly detailed tracing message

Typically, the **Info** level is sufficient for most purposes, although in some cases you may need to log at **Fine** level to diagnose certain issues. **Finer** or **Finest** levels are rarely useful, unless debugging a detailed issue, as they may degrade performance and introduce unnecessary logging that may make it more difficult for diagnosing problems. When running a production Grid that requires very fast performance, you may wish to decrease the level to **Warning** so that only problems are reported.

The log format is: *`{timestamp} {level}: [{component}] {message}`*

Only messages that are at or above the current log level will be logged.

An example of a log message:

```
09/20/05 19:19:10.423 Info: [BrokerServicePlugin] Broker:Total:1
```

Viewing Engine Logs

There are several ways of viewing the logs. The most straightforward is to view the actual log files via the **Log URL List** in the GridServer Administration Tool.

To view an Engine log:

1. In the Administration Tool, click the **Engine** tab, then click the **Engine Admin** page.
2. Find the Engine for which you want to view a log, and from its **Actions** list, select **Log URL List**.

3. A window will open with a list of links for each of the logs residing on that Engine, listed by date and time. Click a link to download and view that log.

You may also wish to view the logs in real time. You can do this via the remote log applet.

To use the remote log applet:

1. In the Administration Tool, click the **Engine** tab, then click the **Engine Admin** page.
2. Find the Engine for which you want to view a log, and from its **Actions** list, select **Remote Log**.
3. An applet window will open, displaying the log on the Engine as events occur. You can click **Clear** to clear the log, or **Snapshot** to capture a screen of the log in a new window.

You can also run the Engine in console mode; typically, this would only be done during development.

Windows: You can run the Engine from a command line with the command `engine.exe -console`. This starts the Engine in console mode, and logging information will scroll on the command window from which it was started.

Unix: You can run the Engine from a command line with the command `engine.sh startfg`. This starts the Engine in the foreground, and logging information will scroll on the terminal from which it was started.

By default, a Unix Engine will detach the stdout from your native Tasklet code. If you wish to see the stdout, set the `DSNODETACHSTDOUT` environment variable in the shell from which you start the Engine for the first time. The variable can be set to any value. Then, the stdout can be found in the `profiles/<engine-name>/logs/engine.out` file.

The **Engine Log Search** page enables you to search for all Severe-level Engine logs for a Service ID across all Engines, and optionally search those results for a keyword. Results are shown with a summary of each matching log for each Engine, with links to corresponding log URLs with excerpts. First, logs are searched for the given Service ID; then they are searched for the regular expression "`.*Severe.*`", then they are optionally searched for a given keyword.

To search Engine logs:

1. In the Administration Tool, click the **Engine** tab, then click the **Engine Log Search** page.
2. Enter a Service Session ID in the **Service ID** box, or select a name from the **Service Name** list. Service Names are provided only when the Reporting Database is available.
3. Enter a keyword in the **Keyword** box, or leave it blank to return all entries.
4. Click **Search**.

Results are shown with a summary of each matching log for each Engine, with links to corresponding log URLs.

Writing to Logs

Your Service will typically also log messages, and you may want these to be logged to the DataSynapse logger.

Java

Both Drivers and Engines capture `stdout` and `stderr`, so typically no changes need to be made to existing implementations to capture logs.

Additionally, the DataSynapse logger is registered as the Apache Commons Logging default handler. If your implementation uses this interface, your messages will be logged automatically. The following is a map of levels to DataSynapse levels:

Commons	DataSynapse
fatal	Severe
error	Severe
warn	Warning
info	Info
debug	Fine
trace	Finer

.NET

The .NET `System.Diagnostics.Trace` facility is used for logging; the DataSynapse logger is simply a Trace listener. The DataSynapse logger will capture any messages written to the Trace facility. This includes .NET Services; any trace message written by the service will be logged to the Engine log.

C++

The `UtilFactory::log` function is the preferred method of logging to the DataSynapse log. It can be used on both the Engine and Driver. If it is necessary to capture native stdout messages on the Engine, there is a hook available from DataSynapse to do so. Note that the logging is only effective after the Driver message server has been instantiated — for instance, after creating a Job or Service object, or after calling `DriverManager::login`.

PDriver

The PDS script language provides redirection of `stdout` and `stderr` to a file, via the `stdout` and `stderr` clauses in the execute statement. For example:

```
execute
       stdout="$DSWORKDIR/pijob.$DSTASKID.out"
       stderr="$DSWORKDIR/error.$DSTASKID"
       ".\resources\win32\lib\PdriverPiCalc.exe $seed $iterations"
```

Writing to the Log directory

The Engine's log directory is always the `[work directory]/log`. Any files written to this directory can be viewed via the Log URL list. This allows you to write log messages to your own files, and view them via the Administration Tool.

The work directory is available as follows:

- Java: The system property `ds.WorkDir`
- .NET: The `System.AppDomain.CurrentDomain` data value `ds.WorkDir`
- C++, Command Service: The environment variable `ds_WorkDir`
- PDriver: The variable `$DSWORKDIR`

Debugging Engines

This section covers the basics on how to attach a debugger to an Engine when necessary to do so. It is intended as a quick aid in getting up and running; it is expected that the developer is familiar with debugging.

Java

The Java Platform Debugger Architecture (JPDA) allows for the connection of a debugger to the Engine via a socket. To open the socket for debugging, add the following to the **Command line-Arguments** parameter in the Engine Configuration of an Engine you wish to debug:

```
-Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,address=[port],suspend=[y/n]
```

`[port]`: The port you wish to open

`[y/n]`: Whether to suspend the process from starting until the debugger is connected. Typically "`n`", as normally you would wait until the Engine logs in and becomes idle, then connect the debugger, and then run the service you wish to debug.

Note that you must only be running a single Engine instance from the Engine Daemon, as an additional instance will not be able to open the same port.

.NET, Windows DLL

Microsoft Visual Studio comes with a remote debugging facility. To debug, you must first make sure that you build with debug symbols, and deploy the symbols (PDB) file with the DLL. Once the engine has logged in, you attach the debugger to the `invoke.exe` process via the **Processes** dialog on the **Debug** menu.

CPPDriver and Linux

GDB can be used to debug native code in CPPDriver or JNI in Linux. Also, GDB can be useful in identifying unusual problems with the Linux JVM. However, there are some subtle issues when trying to use GDB on a JVM, as is the case with the GridServer Engine.

First, when attaching GDB to the Engine, you must specify the `LD_LIBRARY_PATH` to both the Engine components and the JVM components. You must also obtain the process ID of a running `invoke` (or `invokeGCC3`) process from the `ps` command. It's also easier if you run GDB from the base directory of the Engine install (typically `DSEngine`) . The GDB command used is similar to this:

```
LD_LIBRARY_PATH=lib:jre/lib/i386:jre/lib/i386/native_threads:jre/lib/i386/server:resources/lib
/linux gdb bin/invoke $INVOKEPID
```

`bin/invoke` should be replaced with `bin/invokeGCC3` when using GCC3.

This method of running GDB works well for troubleshooting rare JVM problems. However when you are troubleshooting CPPDriver code, a different method should be used. The issue is that CPPDriver loads your application shared objects only when the Tasklet or Service is instantiated, so it becomes difficult to set a breakpoint in the application shared object. (However, more recent versions of GDB feature deferred symbol resolution, which makes this possible.) Further, attaching GDB to a running JVM often has undesired side effects, including halting the JVM depending on the versions of JVM, pthreads, and GDB being used.

The following procedure details how to use GDB with CPPDriver code:

1. Create a GDB initialization file with two commands, one to set the breakpoint and the other to continue. If you take the time to set the breakpoint manually, you risk exceeding some timeouts which will cause the Engine instance to exit. For example, create a file called `yourtest.gdb` containing the following:

   ```
   break YourTest.cpp:42
   cont
   ```

2. Have your Service client call a no-op or initialization method to get the service library (`.so`) loaded. You can call any Service method that doesn't affect the code being debugged. For instance, you could call a method that retrieves the version of the library being debugged, like `getVersion()`. If such a method doesn't exist, you can add something similar to your Service.
3. Attach GDB to the process using the initialization file created in the first step above:

   ```
   gdb -x yourtest.gdb bin/invoke $ENGINEPID
   ```

4. Run the client code used to call the Service method you want to debug.

C++ Compiler Version Notes

Changing the C++ Compiler used with CPPDriver

The CPPDriver and Service bridge libaries are built for nearly all standard compilers used on Windows, Linus, and Solaris. You must link your client application and/or service implementation with the appropriate libraries for the compiler.

You must also run any C++ services against the proper C++ bridge libraries. Typically this is done using Grid Libraries, in that any C++ Grid Library must include the proper bridge Grid Library as a dependency These libraries come already deployed in the `[GS Manager Root]/webapps/livecluster/deploy`.

If you are not using Grid Libraries, you can only use one compiler type for all services per Engine Configuration, and the library must be located in the configuration's Default Library Path. The default libraries are already deployed in the `[GS Manager Root]/webapps/livecluster/deploy`.

Also, because different Linux releases support different compilers which use incompatible versions of the STL, the **GCC Version** property in the Engine Configuration dictates which compiler version of the bridge is supported by the Engine. If using Grid Libraries, you can build your application against all versions you need to support and use the OS element to specificy the proper path of each library. If not using Grid Libraries, you can place all bridge libraries in the `[GS Manager Root]/webapps/livecluster/deploy`.

C++ Multithreading Requirement

Note that all C++ code must be complied multithreaded. This includes both Service and Tasklet code, and Engine or Driver code.

VC++ settings for building Job executable

In order to build a Job executable with Visual C++, you must set the following settings:

1. The **Use run-time library** setting, located in the **Project Setting** dialog box on the **C++** page in the **Code generation** category, must be set to **Multithreaded DLL**.
2. Enable exception handling.
3. Enable Run-Time Type Information(RTTI.)

VC++ settings for building tasklet library

In order to build the Tasklet library with Visual C++, you must set the following settings:

1. The **Use run-time library** setting, located in the **Project Setting** dialog box on the **C++** page in the **Code generation** category, must be set to **Multithreaded DLL**.
2. Define `BUILD_TASKLET_DLL` for the project.
3. Enable exception handling.
4. Enable Run-Time Type Information(RTTI.)

Other C++ Notes

When linking code, you should ensure that your code links with the `dsUtil` library.

Note that when using GCC 3, launching Jobs from an Engine is not supported.

.NET Compiler Notes

The `GridServerNetClient.dll` references `DSJavaNetBridge`, which isn't needed for clients. This may give a build warning about `DSJavaNetBridge`, but does not cause any issues other than the warning message.

.NET Driver Upgrades

As of GridServer 4.0, the .NET Driver (`GridServerNETClient.dll`) is now strongly named. This means that whenever a new version of the .NET Driver is released, via an upgrade, Service Pack or Patch, steps need to be taken for existing clients to allow the assembly to be loaded.

There are two ways of doing this:

- Rebuild the .NET application with the new `GridServerNETClient.dll`.

  or

- Configure the application to allow the new version.

This may be done in various ways, depending on your .NET policy; that is, whether the assembly is deployed into the GAC or used locally, and so on.

An example of the how to do this when the `GridServerNETClient.dll` is used locally, is as follows:

Method 1: Using the "Microsoft .NET Framework 1.1 Configuration" tool

1. To start the tool, click the **Control Panel** menu, click the **Administrative Tools**, then click **Microsoft .NET Framework 1.1 Configuration**.

2. From the **Applications** menu, click **Add an Application To Configure**.
3. If your application is in the list, select it; otherwise, find it using the **Other…** button.
4. Your application is now in the Applications list. Expand your application, and choose **Assembly Dependencies**.
5. Drag the **GridServerNETClient**, noting the version number, to the **Configured Assemblies** icon.
6. Click the **Configured Assemblies** icon. Double-click **GridServerNETClient**, and choose **Binding Policy**.
7. Under **Requested Version**, enter the version you noted in step 5. This is the version that your application was built with. Under **New Version**, enter the new version of the `GridServerNETClient.dll` that have just installed. This allows your application to bind with the new version even though it was built with a previous version.

Method 2: Directly creating the file

In Method 1, the .NET tool creates an Application Configuration file in the directory of the application. However, you may simply create this file yourself.

1. Create a file next to you application executable called *`my.exe`*`.config`, where *`my.exe`* is the name of your executable.
2. Add the following as the file's content:

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="GridServerNETClient" publicKeyToken="42129437978483df" />
        <bindingRedirect oldVersion="4.0.0.12-4.0.0.14" newVersion="4.0.0.15" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Note that the `oldVersion` is the version your application was built with, and the `newVersion` is the version of the new assembly. In this example, `oldVersion` is a range of versions. If your applications already had a configuration file, it should be edited appropriately.

If you have a .NET Service implementation that links to the `GridServerNETClient.dll`, you do not need perform either of these steps. An `invoke.exe.config` file is included in any .NET upgrade that manages this for you. However, you may rebuild your implementation if you wish.

Chapter 3
Creating Services

Introduction

Services provide for remote, parallel execution of code in a way that is scalable, fault-tolerant, dynamic and language-independent. Services can be written in a variety of languages and do not need to be compiled or linked with DataSynapse libraries. There are client-side APIs in Java, C++, COM, and .NET, as well as a Web Services interface. A client written in one language can invoke a Service written in another.

The basic Service execution model is the same as that of other distributed programming solutions: method calls on the client are routed over the network, ultimately resulting in method calls on a remote machine, and return values make the reverse trip. We prefer the term **request** to call or invocation, partly because the operation may be either synchronous or asynchronous.

Services are suitable for implementing parallel processing solutions in which a single computation is split into multiple, independent pieces whose results are combined. This is accomplished by dividing up the problem, submitting the individual requests asynchronously, then combining the results as they arrive. Services also work well for executing multiple, unrelated serial computations in parallel.

Steps in Using a Service

Using a Service involves six steps:

1. **Writing the Service, or adapting existing implementations.** A Service can be virtually any type of implementation: a library (DLL or .so), a .NET assembly, a Java class, a command, script or executable, or even an Excel spreadsheet. A Service does not need to be linked with any DataSynapse libraries, but the remotely callable methods of the Service have to follow certain conventions to enable cross-language execution and to support stateful Services. These conventions will be described below.

   Example code utilizing Services is available in the GridServer SDK. Also, more examples of how to write Services are available in the *GridServer Service-Oriented Integration Tutorial*.

2. **Deploying the Service.** The implementation and other resources required for the Service must be accessible from all Engines. This can be accomplished via a shared file system or GridServer's resource deployment mechanism.
3. **Registering the Service Type**. To make the Service visible to clients, it must be registered as a Service Type in the GridServer Administration Tool.
4. **Creating a Service Session from a Client**. A Client Application is developed that accesses the registered Service Types and creates a Service Session. Each Service Session may have its own state that is client-specific. Because of virtualization, a single Service Session may correspond to more than one physical instance of the Service, such as more than one Engine running the Service's code.
5. **Making requests**. The methods of a Service implementation are called by the Client Application either synchronously or asynchronously.

6. **Destroying the Service Session**. Client Applications should destroy a Service Session when they are done with it.

This chapter describes how to develop a Service Implementation, which will actually run on an Engine. Chapter 4, “Accessing Services” on page 33 describes how to use this Service Implementation from a Client Application.

Service Method Compliance

Although Service methods do not link to DataSynapse libraries, they must comply with a set of rules so that they may be used properly.

Java/.NET Services

- The Service class and all methods called by the client must be public.
- A method may take any number of arguments, and may have a return type of void. The return values of state methods are ignored, and a null will be returned by a Service method with a void return type.
- If the Service is to be used cross-language or accessed by a SOAP client, the arguments and return values must conform to the rules of interoperability, as described in the section “Interoperable Types for XML Serialization and SOAP Clients” on page 29.
- If the Service will only be used by a client of the same language, any serializable object may be used for arguments and return values.
- Overloaded methods are not allowed.
- Methods can throw exceptions within a Service, which will capture and include stack trace data and nested exception data when available.
- In Java, if a Service will be accessed as a Web Service, and a method has a throws clause, it must be throws `Exception`, because the WSDL generator can only handle the `Exception` class. You may still throw any descendant of `Exception`.

C++ Services

- All methods must be public.
- The Service method must either take a `char*`, or a `char**` for multiple arguments. Alternatively, if using the macro it can take a `std:string` or a vector of `std:string`s.
- The method returns data via a `char**` argument, which is set to the returned data. Alternatively, if using the macro it returns a `std:string`.
- Overloaded methods are not allowed.

Command Services

- The name of the method that is called is appended to the command line.
- Argument values are sent to stdin or an input file. If there is more than one argument, the data is separated by the `argDelimiter`, which can be registered on the Service.
- Argument values may instead be appended to the command line, if the option is selected. In this case, they should be passed in as strings by the client.
- If your command spawns subprocesses, it must cancel them if the main command is cancelled.

Client Calling Conventions

Clients must comply with the following rules when calling methods in a Service.

Java/.NET Client

- Arguments are passed into calls as an `Object[]`, which corresponds to the arguments of the method. Note that it MUST be exactly of type `Object[]`. For instance, a set of strings cannot be passed in as a `String[]`. The array length must match the number of arguments.
- For convenience, if the method takes only a single argument, it may be passed directly into the call. It is the equivalent of passing in an `Object[1]` with the 0th element being the object.
- If a method takes no arguments, it can only be called with zero-length `Object[]` or a null object.
- Primitive types are converted to their object equivalents automatically. For example, a Service method that returns a double will return a `Double` on the client.

C++ Client

- Arguments are passed into calls as a `char**`. Alternatively, if using the macro found in `DynamicLibraryFunctions.h`, it can be a vector of `std:string`.
- If a method takes no arguments, it can only be called with a null or zero-length `char*` or `string`.

SOAP Client

SOAP clients are created by using the WSDL that is generated for the Service using the Service Type Registry. The SOAP package that you are using should then create the client proxy when given this WSDL.

Java and .NET Services introspect Services and state methods and create types accordingly, so the proxy methods will reflect the Service Implementation.

C++ and Command services generate WSDL operations for all methods in the same manner. All methods take an `xsd:anyType[]` as an argument, and Service methods return `xsd:anyType`. The input array MUST correspond to an object array of `string` and/or `byte[]`s for the calling language. For instance, if the language is Java, it must be an `Object[]` which contains Strings and/or `byte[]`s. It cannot be a `String[]` or `byte[][]`.

Registering a Service Type

Service Types must be registered in the GridServer Administration Tool on the primary Director, on the **Service Type Registry** page under the **Service** tab. Service Types registered on the Director are then replicated to Brokers. A list of existing Service Types appears on that page, along with a line for adding a new Service Type. Enter the Service Type name on the blank line. Select a Service Implementation, then click **Add**.

In the window that appears after clicking the **Add** button, enter any name, property or option values for the Service Type.

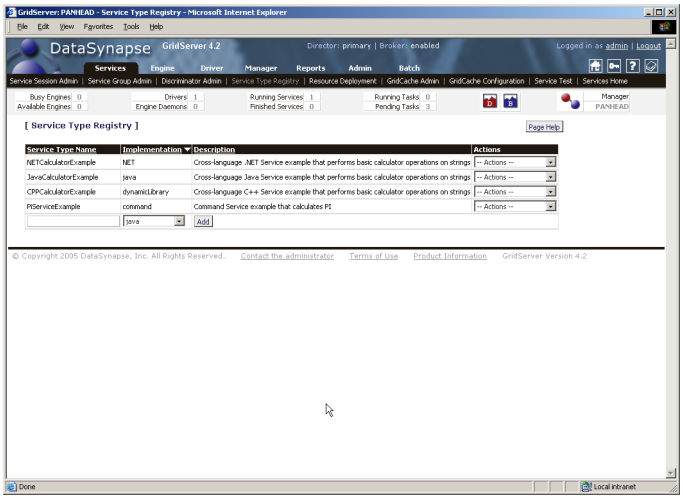


FIGURE 3-1: The **Service Type Registry** page.

Container Binding

Every Service has an associated Container Binding, which binds the Service implementation (the library or command) to the Container of the Service (the Engine). The container binding essentially describes how the implementation is to be used.

The binding contains the following fields:

Field	Description
`initMethod`	Called when a Service Session is first used on an Engine. It is called prior to any requests or updates.
`destroyMethod`	Called when a Service Session is destroyed.
`cancelMethod`	Called when the invocation is cancelled if `KILL_CANCELLED_TASKS` is false for this Service. It is used to interrupt the request if the user does not want the Engine to restart on a cancel
`serviceMethods`	Methods that will perform a request and return a response. These are the actual methods that perform the calculations. The * character can be used to denote all methods that are not bound to any other action.
`appendStateMethods`	Updates state, appending to previous updates.
`setStateMethods`	Updates state, and flushes the list of previous updates.

All methods used must be bound to one of these methods. All methods are optional except `serviceMethods`.

Additionally, the binding also contains the following, currently only applicable to Java and .NET:

Field	Description
`xmlSerialization`	Whether XML serialization is used to serialize objects. See “Interoperable Types for XML Serialization and SOAP Clients” on page 29 for more details.

Field	Description
`TargetPackage`	The package (Java) or namespace (.NET) into which the generated proxy classes will be placed. If not set, the name of the Service is used. Note: When the `targetPackage` name is not set in the **Service Type Registry** page and a generated proxy is used, deserialization errors will occur. To remedy this, edit the **Service Type** on the **Service Type Registry** page of the GridServer Administration Tool and assign a value for the `targetPackage` property.

.NET AppDomains

Services implemented in .NET have full access to .NET's AppDomain functionality, including managing multiple persistent AppDomains across Service invocations, while preserving access to the entire DataSynapse Engine-side API. You can specify an AppDomain as part of a Grid Library deployment and the Engine will set up and manage it automatically.

The **Provider** section in the **Service Type Registry** for .NET Services supports an `appDomainName` value; set this to specify a unique AppDomain for Services created from this Service Type.

When an AppDomain is specified as part of a Service Type definition and an Engine creates the Service for the first time, the Assembly search paths used for the AppDomain depend on how resources are deployed. When a Service uses a Grid Library, the `assembly-path` pathelements for that Grid Library (and any dependent Grid Libraries) are used as the Assembly search paths for the AppDomain.When a Grid Library is not used, the default Assembly search path is used.Once the Assembly search path is determined, the Engine searches it for a valid AppDomain Configuration File, which has the same name as the AppDomain, plus the `.config` suffix.

The `unloadAppDomain` property in the **Service Type Registry** enables you to specify what happens to non-default AppDomains once all Service Sessions using them have been destroyed. Select `true` to unload AppDomains after they are no longer being used.

Language Interoperability

Services provide various levels of interoperability among languages. To provide this interoperability, GridServer can perform conversions on arguments sent to objects. The following describes how arguments are converted between Service Implementations.

Strings and Byte Arrays

All Services can use byte arrays (`byte[]`s) interchangeably with `String`s as arguments. Whenever any conversion is performed, it is done using UTF-8 encoding. For example, if an argument is of type `String`, and the client passes in a `byte[]`, the `byte[]` will be UTF-8 encoded and passed into the method as a `String`.

Because a C++ Service always returns a `string/char*`, the returned type of an invocation must be converted to a `String` or `byte[]`. The type of conversion made is based on the first argument to the invocation. If a string is passed in as the first argument, it will return a string, and if it is passed a `byte[]`, it will convert the string to a `byte[]`. If there are no arguments, it will return a `byte[]`. This is most relevant for a .NET or SOAP client, as string I/O must be ASCII. If you are returning binary data, make sure that the first argument is a `byte[]`.

A Command Service will only convert the output data to a `String` if the first argument is a `String`, and the `appendArgsToCommandline` option is false.

Java and .NET Services do not convert return values if they are `String`s or `byte[]`s.

Object Conversion from Strings and Byte Arrays

Java and .NET Services will automatically attempt to convert `String/byte[]`s to and from `Object`s when necessary. This can be useful when calling these Services from a different language, or when using Service Runners from Batches.

If an argument is not a `String` or `byte[]`, and it is passed in as such, an attempt will be made to convert it. If the data is a `byte[]`, it will first be converted to a `String`. Then the `String` will be converted to the `Object` as follows:

Input Argument	String Argument-to-Object Conversion
Primitives	The primitive wrapper class's parse method
Date, Calendar (Java)	`DateFormat.getDateTimeInstance().parse`
DateTime (.NET)	`DateTime.Parse`
org.w3c.dom.Document (Java)	Uses the parse method from the DocumentBuilder given by `DocumentBuilderFactory.newInstance().newDocumentBuilder()`
XmlDocument (.NET)	`XmlDocument.loadXml`
Other	If the class has a constructor that takes a single `String` as an argument, it will use that constructor.

If the return value is not a `String/byte[]`, and the client is not of the same language as the Service, the returned value will be converted to a `String`, as follows:

Return Type	Returned Object-to-String Conversion
Primitives	The object-equivalent `toString()` method
Date, Calendar (Java)	`DateFormat.getDateTimeInstance().format`
DateTime (.NET)	`date.ToUniversalTime().ToString("r", DateTimeFormatInfo.InvariantInfo)`
org.w3c.dom.Document (Java)	The transform method from the Transformer given by `TransformerFactory.newInstance().newTransformer()`
XmlDocument (.NET)	`doc.WriteTo(XmlTextWriter)`
Other	The `toString` method is used.

XML Serialization for Java and .NET

XML serialization provides the following features

- It allows Java and .NET to use rich objects as arguments and return values with each other.
- It allows a client to use a Service with such objects, without needing the original implementation classes. This is because client-side proxy classes are generated.

To use XML serialization, it must be enabled on the Service Type. Note that by enabling this, the other interoperability conversions are no longer used. Additionally, the client must use the proxy that is generated using the Service Type Registry, which contains all user-defined types.

The arguments and return values on such services must be Interoperable Types, as discussed in the following section.

Interoperable Types for XML Serialization and SOAP Clients

When using XML Serialization, or when such a Service will be accessed via the Web Service interface, the parameters and return types must be interoperable, or interop, types. These types are the generally accepted SOAP interop types, and are as follows:

Type	Description
Primitives	`byte`, `byte[]`, `double`, `float`, `short`, `int`, `long`, `string`, or `Calendar` (Java) or `DateTime` (.NET).
Arrays	The type may be an array of any interop type.
User-Defined Types	User-defined types can be used, as long as they follow the standard "bean" pattern. For both languages, all data must be interoperable types (including other user- defined types.) For Java, all data must be Java Bean properties; that is, each must have public get/set methods. For .NET, all data must be public fields. When generating a proxy for this Service type, user-defined types will result in generated classes. You may have other data and methods in the type, such as private non- interop fields, but they will be ignored and not reflected in the generated class. Also, user-defined types must be concrete; abstract classes and interfaces are not allowed.
Data References	This type can be used as an argument, return type, or GridCache object, and is interoperable whether XML Serialization is on or off. For more information on Data References, see "Data References" on page 39.

The following is an example of a Java Interop type:

Example 3.1: Java Interop Type Example

```
public class Valuation {
    private java.util.Calendar valuationDate;
    private double value;
    private MarketData

    public Valuation() {
    }

    public java.util.Calendar getValuationDate() {
        return valuationDate;
    }
```

Example 3.1: Java Interop Type Example (Continued)

```
        public void setValuationDate(java.util.Calendar valuationDate) {
            this.valuationDate = valuationDate;
        }

        public double getValue() {
            return value;
        }

        public void setValue(double value) {
            this.value = value;
        }
    }
```

The following is an example of a .NET Interop type:

Example 3.2: .NET Interop Type Example

```
    [Serializable]
       public class Valuation {
           public DateTime valuationDate;
           public double value;
       }
```

Maintaining State

To maintain state on a Service, you would typically use a field or set of fields in your object to maintain that state. (For C++ or Command Services, state is saved in a slightly different manner.) Because a Service Session can be virtualized on a number of Engines, adjusting a field's value using a Service request will only adjust that value on the Engine that processed that request. Instead, you must declare the appropriate class method as a stateful method in the Service Type Registry, and use the `updateState` method to guarantee that all Engines will update the state. All methods that are used to update the state must be registered as such on the Service type, either as one of the `setStateMethod`s or `appendStateMethod`s.

When an Engine processes a Service request, it first processes all update state calls that it has not yet processed, in the order in which they were called on the Service instance. These calls are made prior to the execution of the request. The append value is used to determine whether previous update calls should be made. If append is false (a "set"), all previous update calls are ignored. If append is true, all calls starting from the last "set" call will be performed. Typically, then, "append" calls would be used to update a subset of the state, whereas a "set" call would refresh the entire state. If your Service instance is intended to be a long running state with frequent updates, you should on a regular basis use a "set" call so that Engines just coming online do not need to perform a long set of updates the first time they work on this Service instance.

Initialization

The `initMethod`, one of the container binding fields defined above, is typically used to initialize the state on a Service that maintains state. It may be also used for other purposes, such as establishing a database connection. The `initMethod` is called with the initialization data the first time an Engine processes a request on a Service instance. It will also be called prior to an `updateState` call if it has not already been called.

Cancellation

A request may be cancelled for a number of reasons. It can be directly cancelled by the Admin interface or Administration Tool, or it will be cancelled if the Service Session is cancelled. If the `killCancelledTasks` option is true for this Service, the Engine process will simply exit, and the Engine will restart. However, in many cases it is not necessary to do so, and you would prefer to simply interrupt the calculation so that the Engine becomes immediately available.

In this case, the `killCancelledTasks` option should be false, and a `cancelMethod` should be implemented and registered on this Service type. This method must interrupt any Service method that is in process. It is also possible for the `cancelMethod` to be called after the Service method has finished processing, so the implementor must take this into account.

If a request is cancelled due to the Service being cancelled, the `cancelMethod` will be called prior to the `destroyMethod`.

Destruction

Often times a Service will need to perform some cleanup on the Engine when the instance is destroyed, such as closing a database connection. If so, a `destroyMethod` should be implemented and registered on the Service type. This method will be called whenever a Service instance is destroyed. It will also be called on any active Service Sessions on an Engine whenever an Engine shuts down.

Service Instance Caching

Engines maintain a cache of all Engine Service Instances that are currently active on that Engine, set by the Engine Configuration. If an Engine is working on too many Sessions, Engine Service Instances may be pushed out of the cache. In this case, the `destroyMethod` is called, and it is as if the Engine has not yet worked on that Service. That is, if it processes a subsequent request, it will initialize and update as if it were the first time it worked on that Service.

Invocation Variables

While a Service implementation can be completely independent of DataSynapse libraries, there are certain occasions on which you may need to interact with the GridServer environment on the Engine. This is accomplished via variables that are retrieved in various ways dependent on the type of Service:

Java: System properties

DynamicLibrary: Environment variables, with the same name as Java variables, except with dots replaced with underscores. You can also use symbolic constants provided by `DynamicLibraryFunctions.h`.

.NET: System.AppDomain.CurrentDomain data values

Command: Environment variables, with the same name as Java variables, except with the dots replaced with underscores

The Engine provides the following variables:

Variable	Description
`ds.ServiceSessionID`	The unique identifier for the Service Session being invoked.

Variable	Description
`ds.ServiceInvocationID`	A number uniquely identifying the invocation (task) of the Service instance.
`ds.ServiceCheckpointDir`	The directory that the Service should use for reading and writing of checkpoint data, if checkpointing is enabled for the Service.
`ds.ServiceInfo`	If this variable is set in an invocation, the value will be displayed in the Task Admin upon completion of the invocation.
`ds.WorkDir`	The work directory for the Engine. This variable is set to the directory from which the Service is executed; by default, it is `./work/`*`machinename-instance`*, relative to the Engine installation directory, where *`machinename`* is the name of the machine running the Engine, and *`instance`* is the number of the Engine instance. For example, a single Engine machine will have a *`machinename`*`-0` directory; one with two Engine instances will also have a *`machinename`*`-1` directory. In each Engine instance directory, there is a `log` directory containing Engine logs, and a `tmp` directory. Note that the `tmp` directory is periodically deleted by the Engine. The **Temp File Time-to-Live (hours)** setting in each Engine Configuration controls the frequency with which the Engine cleans this directory.
`ds.DataDir`	The data directory for the Engine, which is the directory in which DDT data is stored. The cleanup frequency is also controlled by the Engine Configuration.
`ds.GridLibraryPath`	A path made by concatenating the root directories of all the expanded and loaded Grid Libraries.

In addition, any environment variables available on the Engine will also be available in the Engine Service Instance.

Chapter 4
Accessing Services

Introduction

This chapter provides a detailed description of how to access and utilize Services with GridServer, with three different methods:

The Service API — A GridServer Driver provides an interface between a client application written in Java, C++, .NET, or COM. It provides methods and an API that can be used to develop applications that can then access Services.

Web Services — Typically, you would use Web Service interface when your client cannot use one of the GridServer Drivers, such as if your client code is written in a language other than C++, Java, .NET, or COM. Also, you may prefer to use this interface if you are standardized on Web Services and you are already utilizing a rich SOAP client toolkit.

Proxy — GridServer can automatically generate of a client proxy class that mirrors the registered Service Type. This proxy generation mimics the WSDL proxy generation of a Web Service; the difference is that the proxy makes its calls via a Service object on a Driver rather than using SOAP over HTTP.

The Service API

Services can be accessed by using the Service API in Java, C++, .NET or COM to develop an application. Each of these Drivers contain API documentation describing how to do this. For example, when using Java, refer to the Javadocs found in the GridServer Administration Tool, for the package `com.datasynapse.gridserver.client`. The `Service` class is used to access the Service either synchronously or asynchronously.

Further examples of developing applications using the GridServer APIs can be found in the *GridServer Service-Oriented Integration Tutorial*.

Web Services

The **Web Services** interface provides a mechanism for a client to create and use Service Sessions, without the need for a Driver. The Service can be used by a client that is implemented in a language that supports Web Services using SOAP over HTTP. Web Services hosted by GridServer can also be stateful, like Services, and support Service options.

Any registered Service type is automatically exposed as a Web Service. For Java and .NET, arguments and return values must be standard Web Service interop types. See Chapter 3, “Creating Services” on page 23 for more details.

Service Routing

The Director has a `DriverAdmin` Web Service, which has a method called `getServicesURL(String serviceName)`, which returns the URL of the Service on a Broker suitable for the Driver Profile associated with the user. This Broker is chosen in the same manner a Broker is chosen for a DataSynapse Driver. Alternatively, a client can select a specific Broker with which to route SOAP requests directly.

Web Service Functionality

The WSDL for this Service is obtained by using the **Service Registry** page in the GridServer Administration Tool, or at a URL which has the following form:

`http://`*`host`*`:`*`port`*`/livecluster/services/`*`ServiceName`*`?WSDL`

The `serviceURL` can be obtained from the `DriverAdmin` Web Service on a Director, as mentioned above.

A default Service instance is always available at the `serviceURL`. Any methods designated as service methods in the binding can be run as RPCs. These calls will be executed on any available Engine.

The default Service instance is created when a Service is accessed like a Web Service. This same Service instance will be used by any other attempts to access the Service like a Web Service until it times out; a later attempt to similarly access the Web Service will cause it to be created again. This process is transparent to the user. However, if you wish to use state with a Web Service, you must initialize it differently. This is covered in "Service Instance Creation/Destruction", below.

In this way, the Web Service behaves just like any deployed Web Service, in that the Service provides WSDL and processes SOAP RPCs. The difference is that these RPCs are distributed to Engines and executed in parallel, rather than serially as in a typical Web Service provider.

The WSDL generated from Java and .NET uses introspection to generate the exact argument and return types. For dynamic library and command services, the argument type will always be an "array of anything" (which really has to be an array of strings or byte arrays) and the return type is anything. Furthermore, the methods generated in the WSDL are the ones specified in the Service Type Registry and no others (since there is no introspection), whereas in the Java and .NET cases you can write `"*"` for the list of service methods, and it will find all public methods.

The endpoint in the WSDL returned will point to the Broker from which you generated the WSDL. To get proper Broker routing, you have to go through the Director using the `ServiceManager` Web Service in order to get a Broker URL and assign that URL to your client's proxy.

Advanced Functionality

Management of Service instances, state, and asynchronous submission/collection is also provided, and handled by the use of the `SOAPAction` attribute. The appropriate attribute, based on the container binding, is automatically attached to the corresponding operation in the WSDL.

NOTE: In order to take advantage of this, your SOAP client must maintain its session.

Service Instance Creation/Destruction

Because the Web Services specification does not account for how to create and manage stateful Web Services, the init and destroy methods (that is, the operations associated with `initMethod` and `destroyMethod` in the container binding) of a Web Service will behave differently when hosted by GridServer. The operation associated with the `initMethod` in the container binding is used to create a new Service Session. If the Service does not have an `init` method, a default `create` operation is added automatically. This session is independent of the default session and any other sessions of this Service Type. It is essentially the same as using the Service Factory to create a new session using the API. The return value from the operation is the URL of this new session, and will be of the form `[service URL]/[id]`, where `id` is the Service ID. For example (on a proxy generated in C#):

```
// create the Service
ExampleService service = new ExampleService();
// create a stateful Service instance by calling the initMethod,
// and assign the URL to the new instance's URL.
service.Url = service.create(...);
```

Calling the `create` method on the proxy does not directly call the corresponding method on the Service. Instead, you have to reset the proxy's endpoint to the endpoint of the new Service session as above. When you make the first method invocation on the new session, that Service's `create` method will be called prior to the first method invocation. The method of changing your endpoint may vary depending on what client you are using (gSoap, Axis, and so forth.)

A `destroy` operation is also added, which destroys the session. If you have a `destroyMethod` registered, the operation will be that method. Otherwise, a `destroy` operation is added.

Asynchronous Submission

Every method registered as a `serviceMethod` has an additional asynchronous operation created for it. The name is `[method name]_Async`, and the return value is the ID (string) of the invocation, which can be used for collection.

An additional operation is provided called `collectAsync`. This method takes a single ID (string) argument as its input. If the value of the argument is null, it collects the next result, otherwise it collects the result for that ID. The operation returns two values. The first is an invocation result value, and the second is the ID of the invocation. If the ID is null, the next available invocation result value is returned; otherwise, the value for the provided ID is returned. The result of a `collectAsync` call can be one of five states:

- The returned ID is non-null. In this case, the result value is the result for that ID.
- The returned ID is null, but the result is a long. In this case, it is the amount of time, in msecs, that the client should wait before polling again.
- Both the ID and result are null. This means that there are no outstanding results to collect.
- A SOAP Fault is returned, of type `client`. This means that a request failed. The SOAP Fault `Actor` is set to the ID, and the `detail` contains the exception.
- A SOAP Fault is returned of the type `Server`. This means that the instance failed. The `detail` contains the exception.

Typically, your client would implement a collector thread that continually polls the `collectAsync` method to gather output data as it becomes available.

Note: Most web service client generators handle multiple return values via a wrapper class (such as Axis) or passing the additional argument by reference (such as C#.)

State Updates

Methods registered as state update calls are simply marked with the appropriate SOAPAction. When called, they perform an update just as if from the API. State updates may only be done on non-default service instances

Fault Handling

Any execution exception is handled by returning a standard SOAP fault.

- **faultCode**: If the Service implementation had an exception, the code will be `Client`. It will be `Server` in any other case.
- **faultString**: The exception message and stack trace.

If the Web Service throws an exception on an asynchronous Service, the fault will be returned on the collection request.

Authentication

Authentication is performed if Driver Authentication is enabled on the Director. To enable Driver Authentication, see “Enabling Client Authentication” on page 70 of the *GridServer Administration Guide*. Basic HTTP Authentication can be used, which is supported by most SOAP clients, such as the .NET `SoapHttpClientProtocol` class, and Apache Axis client.

Here are the steps to deploy a Web Service for a SOAP client using Driver authentication.

Java with Axis:

1. Deploy the Web Service.
2. Generate the Java Proxy classes as follows:

```
java org.apache.axis.wsdl.WSDL2Java
http://example.com:8000/livecluster/services/JavaDealValuatorExample?wsdl
```

3.) Create an instance of the proxy as follows:

```
JavaDealValuatorProxy proxy = (new
JavaDealValuatorServiceLocator()).getJavaDealValuatorProxy();
((org.apache.axis.client.Stub)proxy).setMaintainSession(true); // maintain session
```

4.) When using Driver Authentication, do the following:

```
((org.apache.axis.client.Stub)proxy).setUsername("your username");
((org.apache.axis.client.Stub)proxy).setPassword("your password");
```

.NET:

1. Deploy the Web Service.
2. Using a .NET cmd shell, do the following:

```
wsdl.exe http://example.com:8000/livecluster/services/JavaDealValuatorExample?wsdl
```

3. Create an instance of the proxy as follows:

```
using System;
```

```
using System.Net;
using System.Web.Services.Protocols;

JavaDealValuatorService proxy = new JavaDealValuatorService();
proxy.CookieContainer = new System.Net.CookieContainer // maintain state
```

4. When using Driver Authentication, do the following:

```
proxy.credentials = new NetworkCredential("your username", "your password");
```

Proxy Generation and Services as a Web Service Binding

Proxy Generation is the automatic generation of a client proxy class that mirrors the registered Service type. This proxy generation mimics the WSDL proxy generation of a Web Service; the difference is that the proxy makes its calls via a Service object on a Driver rather than using SOAP over HTTP.

Essentially, the Service can be thought of as a binding to a virtualized Web Service that can process asynchronous requests in parallel. Additionally, because the proxy does not expose any DataSynapse classes, it provides a standards-compliant approach to integrating applications in a vendor non-specific way.

The following rules apply to the generated proxy:

- The use of the proxy class is completely independent of the DataSynapse API. That is, client code that uses the proxy class does not need to import or reference any DataSynapse classes.
- If there is an `initMethod`, the proxy constructor takes any arguments to that method.
- All service methods produce synchronous and asynchronous versions of the method on the proxy.
- Each update method has a corresponding update method on the proxy.
- Since the `cancelMethod` and `destroyMethod` are called implicitly, they do not generate methods on the proxy.
- The `targetPackage` field indicates the package (in Java) or namespace (in .NET) in which the generated classes are placed. If not set, it is the name of the Service.
- If xmlSerialization is used, classes are generated for all non-primitive types, which must be interop types. If not, they can be any serializable type, they are not generated, and the client must have access to those same classes (via a JAR/Assembly.)
- When generating proxies, a Java `Calendar` object will be represented by a `DateTime` object in a .NET proxy, and vice-versa.

The proxy is generated using the **Service Type Registry** page. The proxy is generated on an Engine, so an idle Engine must be available for the generation to succeed.

Java Proxy Example

The following is an example of methods from a Java generated proxy:

Example 4.1: Java Generated Proxy Example

```
public class JavaDealValuatorProxy extends ServiceBindingStub {

public JavaDealValuatorProxy(examples.dealvaluator.client.java.MarketData data) throws
Exception {...}
```

Example 4.1: Java Generated Proxy Example (Continued)

```
    public void destroy() {
           super.destroy();
       }
   }
```

.NET Proxy Example

The following is an example of a .NET proxy.

Example 4.2: .NET Generated Proxy Example

```
namespace examples.dealvaluator.client.net {
public NETDealValuatorProxy(MarketData data) : base("NETDealValuatorExample",
new object[] {data}, null, null, true) {}
...

new public void Destroy() {
            base.Destroy();
       }
}
```

Service Options

Each Service is equipped with a Options object, which contains various configuration parameters and settings. For example, some commonly used options include `PRIORITY` and `GRID_LIBRARY`. A complete list of all options available for the `Options` object is available in the API reference documentation.

Service Options can be set in two ways: in the Service Type Registry, or when creating the Service Session with the client. If an option is set in the registry, it cannot be overridden by the client. If it is left as `[not set]` in the registry, and it is not set by the client, the default value is used.

Service Session Context

The `ServiceSessionContext` is a convenience class for accessing information about an invocation, such as the session and task IDs, while it is running on an Engine. This is an alternative to using, for example, the system properties when running a Java Service. Using this class allows for immediate updating of invocation information; setting the `INVOCATION_INFO` system property only updates at the end of the invocation.

The `ServiceSessionContext` object can be reused; the method calls will always apply to the currently executing Service Session and invocation. All method calls should be made by a `service`, `update`, or `init` method; if not, it may throw an `IllegalStateException` or return invalid data. For instance, if a Service method spawns another thread which uses this class, the context is only valid until the Service method returns.

Shared Services

A Shared Service is an instance of a Service that is shared by Clients executing on different processes or machines. When a Service is Shared, it means that the state of the Service is maintained across the Client boundary. A Globally Shared Service Instance is a shared Service that can have only one instance for the Manager.

A Shared Service is created when a Service is created with the `SHARED_SERVICE_NAME` option specified. Any client of the same type attempting to create a Service with the same name and the same Shared Service name will attach to the already created Service. All clients sharing the shared instance will share the same Service ID. Clients must be of the same type to share Services (for example, a Java Driver cannot share a Service created with CPPDriver.) You cannot use any init data when you create a Shared Service.

When the client cancels the Service instance, it only cancels the tasks submitted by that client. The Service can only be cancelled by the admin page or Web Service Admin interface. The Shared Service becomes inactive when the last client detaches from the Service. The Service will be closed if the `SHARED_SERVICE_INACTIVITY_KEEP_ALIVE_TIME` is 0 or not set. Otherwise, it waits that amount of time before closing. Also, note that when using Shared Services, Engine state is not maintained in failover, unless all Drivers that updated state are still running the instance.

Service Groups

Service Sessions can be collected together in a group to aid in administrative tasks. A convenience class called `ServiceGroup` is provided in the API, which allows you to create a Service Group and later create new Service Sessions within the Service Group. Each new Service Session created within a Service Group will be automatically assigned `Description.SERVICE_GROUP_ID`, a generated random unique ID for that group.

In the GridServer Administration Tool, you can view and maintain Service Groups in the **Service Group Admin** page on the **Services** tab. The **Service Group Admin** page enables you to take actions on an entire group of Services at once, similar to the way you can act on Services on the **Service Admin Page**. For example, you could cancel a Service Group, which would cancel all of the Service Sessions within that Service Group.

Data References

Data References are a convenient programming interface for passing lightweight references to data across the network. A Grid client or service can create Data References, pass them over the network, but leave the data where the original Data Reference was created. If any Grid client or Grid node actually needs the data, it can de-reference the object and the data will automatically be downloaded from the original source.

This abstraction can be used for generalizing Grid workflows. A Grid client can receive the results of a particular service as a reference, and then send another request to the Grid with that reference. The GridServer Engine that services the request will de-reference the data object, loading it from the original Grid node that produced the data. This is equivalent to passing pointers across the network.

A `DataReference` is an Object that can be passed interoperably if it is used as an argument, return type, or a GridCache object. Note that it must be the actual object passed to work interoperably; it can't be part of another object.

A `DataReference` is created using a `DataReferenceFactory`. Data References cannot be created with a null source. The `fetch` methods are use to retrieve the actual data. The data is not cached after a `fetch`, that is, any time a `fetch` is performed, the data is retrieved. If the reference has expired, a FileNotFound exception is thrown on a `fetch` If the client is down, a Connect exception is thrown, although in some cases connections can be refused due to other reasons, such as socket backlog limitations.

C++ Data References

Because there is no inherent serialization support in C++, additional functionality is provided to convert a Data Reference to a `byte[]` (and the reverse) so that it may be sent to another client and used by that client.

Also, because reflection is not available in C++, object Data References are not available via the C++ API.

Service Collection

By default, the Driver Service Instance will immediately collect results as soon as they are available. In some cases, you may need to either never collect the results, or collect them at a later time. Values used with `COLLECTION_TYPE` include `IMMEDIATELY`, `LATER`, and `NEVER`.Values used with `COLLECTION_TYPE` are as follows:

IMMEDIATELY Task outputs are collected as soon as they are ready.

LATER Task outputs are not collected in this Service, another Service will collect them later. See "Deferred Collection", below, for more information. This is not available from a Service proxy.

NEVER Task outputs will not be collected. This may be used, for example, in the case where a Service may write data directly to a database. This is not available from a Service proxy.

The `LATER` and `NEVER` collection modes are not for long running services. They should be used only for batch submissions that finish quickly. If used for Services with indefinite duration, there will be no way to clean up the inputs.

Deferred Collection

The `LATER` mode indicates that the client can submit and update data to the Service Session. It will not collect the results, however, as another instance will attach to the Service Session to collect the results. None of the results are removed from the Service Session until it is destroyed by the collector. The `LATER` mode cannot be used from a Service proxy.

There are two reasons then to use this method:

1. To speed the submission of requests to a Service Session so that Engines can start working as soon as possible. This is because the collection of data does not start until submission is complete.
2. To recover from a failure in the application that embeds the Driver. Since results are not removed until the Session is destroyed, if the application undergoes a failure it can recollect the results when it restarts.

Deferred collection Services require that the submitting Driver to call destroy on the Service to indicate that submission is complete. If you are using the submitting Driver in such a way that it exits after submitting the tasks and calling destroy, keep in mind that you should not call `System.exit` or `exit` from the `ServiceLifecycleHandler`, as the destroy message will never get to the Broker. Also note that if you are exiting the submitting Driver immediately, you must set `DirectDataTransfer` to false in the `driver.properties` file

After creating a Service with deferred collection, `ServiceFactory.getService()` can be used to retrieve results. When all results have been collected, call destroy to indicate to the Broker the instance has collected all outputs and the Session should be destroyed. Multiple collectors can be created, but keep in mind that if a collector calls `destroy()`, the Service will be destroyed and no other collectors will be able to finish collecting outputs.

The following is an example of how to use the LATER mode with recovery:

Example 4.3: Deferred Service Collection

```
//
    // Creates a new Session and submits requests to the Session
    // @param serviceType The type
    // @param methods The list of methods
    // @param args The list of arguments
    // @return The id of the Session
    // @throws Exception on error.
    //

        private String submitService(String serviceType, String methods[],
Object[][] args) throws Exception {
       // create the session as a Collection.LATER type
        Properties props = new Properties();
        props.setProperty(Options.COLLECTION_TYPE, Options.Collection.LATER);
      Service cs = ServiceFactory.getInstance().createService(serviceType, null,
props, null);
       // Submit all requests.
      // Note that the handler must be null because this Instance cannot collect.
        for (int i = 0; i < args.length; i++) {
            cs.submit(methods[i], args[i], null);
        }
        String id = cs.getId();

 // destroy to indicate that submission is complete, and to free local resources
        cs.destroy();

        // now save this ID to a file, for recovery purposes
        saveServiceForRecovery(id);
        return id;
    }
//
    // Starts collection of results from a Collection.LATER Session
    // @param id The id of the session
    // @param handler The invocation handler
    // @throws Exception on error.
    //

 private void collectService(final String id, ServiceInvocationHandler handler)
throws Exception {
       // create a handler that removes this service id from the list in the file
when it is finished
        ServiceLifecycleHandler slc = new ServiceLifecycleHandler() {
            public void destroyed() {
                removeServiceFromRecovery(id);
            }
            public void destroyed(ServiceException e) {
                removeServiceFromRecovery(id);
            }
        };
```

Example 4.3: Deferred Service Collection (Continued)

```
          // get an instance of the session, which starts collecting results
          Service cs = ServiceFactory.getInstance().getService(id, handler, slc);

          // set the service to be destroyed when it finishes collecting all output
          cs.destroyWhenInactive();
      }

      //
      // Runs a service by first creating a Collection.LATER Session, submitting
all requests,
      // then getting the collection instance to collect the results.
      // @param serviceType The type
      // @param methods The list of methods
      // @param args The list of arguments
      // @param handler The invocation handler
      // @throws Exception on error.


   //
     private void runService(String serviceType, String methods[], Object[][] args,
ServiceInvocationHandler handler) throws Exception{
          String id = submitService(serviceType, methods, args);
          collectService(id, handler);
      }


//
    // Recovers from an application failure by starting collection of Sessions that
      // did not complete collection prior to failure
      // @param handler The invocation handler
      // @throws Exception on error.
      //
      private void recoverAll(ServiceInvocationHandler handler) throws Exception {
          String[] recovered = getAllRecoveryServices();
          for (int i = 0; i < recovered.length; i++) {
              collectService(recovered[i], handler);
          }
      }
```

No Collection Service

The API allows a `NEVER` collection mode, which allows a Service to submit tasks and not collect them. Such a Service may, for example, write results to a database. Services created with `NEVER` collection can only submit and update. Calls to execute throw an Exception. This collection mode is not available when using a Service proxy.

A `NEVER` collection Service is created by setting the `CollectionType` option to `NEVER`.

Calling destroy will release resources locally on the Driver, and indicate that the Instance is finished with submission. If the Driver is shut down and times out, the session will be considered to be done submitting, as if the Driver had called destroy. When the session is finished submitting due to one of the two prior actions, and all tasks have completed/failed, the session is automatically closed.

Engine Pinning

Engine Pinning enables a Service Session to specify that once an Engine has worked on a Service Session, it will not work on any other Service Sessions as long as that session is in progress. This enables you to quickly replicate 1-to-1 architectures, or if for legacy reasons or lack of availability to source code to maintain massive amounts of expensive-to-replay state on an Engine.

Engine Pinning is typically used in conjunction with Max Engines to limit one or a set of Engines to work solely on a Service.

This feature is available for any type of Service. It is exposed as a two Engine-side methods, `pinToService()` and `unpinFromService()`. If an Engine calls `pinToService()` while working on a task for a Service, it will be marked as pinned to the session when it completes the task, regardless of whether the task was successful. From this point, the Engine will only take tasks from this Service. Once the session is finished, or when then Engine calls `unpinFromService()`, the Engine is no longer pinned and can work on other Services.

Engines that are pinned will be unpinned automatically in the following circumstances: if the Broker loses its connection with the Engine for any reason (such as loss of heartbeat), if the Engine process terminates for any reason, if an Engine performs a soft logoff due to Engine Sharing, if a task fails due to an error in DataSynapse code (such as reading input), or if the session is destroyed.

Exceptions issued by user code that result in task failure do not cause an Engine to be unpinned, unless the exception specifies Engine restart, in which case the above requirement applies.

Other pin/unpin strategies—for instance, unpinning after a certain amount of idle time, or when the Broker queue is empty—can be implemented as a separate Engine thread that polls for a given condition and unpins as necessary.

Chapter 5
The Tasklet API

Introduction

The Tasklet API is available in both C++, using the CPPDriver, and in Java, using the JDriver. It is suitable for use when your application code on the Driver and the Engine side is written both in Java or both in C++. It is the forerunner to the more loosely-coupled Service model.

Note: This material is also covered with code samples in the *GridServer Object-Oriented Integration Tutorial*. This chapter is designed for readers who want a conceptual reference to the API.

The Tasklet API

The Tasklet API consists of four types of objects: `Tasklet`, `TaskInput`, `TaskOutput`, and `Job`.

A **Tasklet** is the object that is created on the Engine side. It packages the computation's common data and behavior needed to run one unit of work in the overall problem being distributed. Like a Servlet, a `Tasklet` contains a method for doing the work, as well as other methods for lifecycle management. A `Tasklet` takes a `TaskInput` as input, operates on it, and produces a `TaskOutput` as output. A `TaskInput` packages the data and code that is unique to one work unit in the overall computation, and the `TaskOutput` packages the results of an individual unit of work. Although it is helpful to think of a task as a combination of a `Tasklet` and one `TaskInput` producing one `TaskOutput`, there is no `Task` object in the Tasklet API.

A `Job` object, represents the overall group of work being computed. The Job is the coordinator of the individual work units or tasks. Using a Job object, your application creates a Job specific Tasklet, submits `TaskInputs`, and processes the `TaskOutputs` as they arrive. The Tasklet API code provides access to the GridServer Driver code compiled with and running within your application. The Driver collaborates with the GridServer Manager to schedule and manage Jobs, distribute tasks, and provide the execution environment that guarantees GridServer fault tolerance.

TaskInput and TaskOutput

TaskInput and **TaskOutput** are marker interfaces and contain no methods. Their purpose is to provide type safety for objects as valid GridServer Tasklet API objects and ensure that the objects can be serialized for transmission across the network as part of the distributed calculation managed by GridServer.

In Java, `TaskInput and TaskOutput` extend `java.io.Serializable`.

In C++, `TaskInput and TaskOutput` extend the incorporated class `Serializable`. Since object serialization is not a built-in feature of the C++ language, this class provides the mechanism by which the C++ application code and the GridServer middleware exchange object data. It contains two pure virtual methods, `read` and `write`, that must be implemented in any class that derives from it.

Tasklet

The most important `Tasklet` method is `service`. It implements the computation that is to be performed in parallel. The `service` method takes a `TaskInput` as argument and returns a `TaskOutput`.

Like `TaskInput` and `TaskOutput`, the Java `Tasklet` class extends `java.io.Serializable`. This means that the `Tasklet` objects may contain one-time initialization data, which need only be transferred to each Engine once to support any Task from the same Job. (The relationship between `Tasklet`s and `TaskInput/TaskOutput` pairs is one-to-many.) In particular, for maximum efficiency, shared input data should be placed in the `Tasklet`, and only data that varies across invocations should be placed in the `TaskInput`s. `Tasklet` data can be changed while the job is running via the `update` method.

Job

The `Job` object specifies methods that are implemented by the client application and are called by the GridServer Driver in order to coordinate Job execution. These callbacks to notify the client application code when tasks complete, when the Job is completed, or when errors occur. A `Job` has a single `Tasklet`, and vice versa. In addition, the `Job` defines static methods for instantiating `Job` objects based on XML configuration scripts.

The Job should:

- Specify which `Tasklet` is associated with the `Job`, by calling the `setTasklet` method,
- Provide the `TaskInput`s for the `Job`, by calling the `addTaskInput` method from within the `createTaskInputs` method
- Start the `Job`, by calling `execute` or `start`, and
- Process `TaskOutput` results in the `taskCompleted` method.

In addition, for C++ implementations, the Job must specify the library that contains the Tasklet implementation to be shipped to the remote Engines with a method called `getLibraryName`.

Both the C++ and Java versions of the API provide blocking (`execute`) and non-blocking (`start`) Job execution methods.

When you write a `Job` class, you should minimally write the following methods:

- A constructor that can accept parameters for the `Job`.
- A `createTaskInputs` method to create all of the `TaskInput` objects. Call the `addTaskInput` method on each `TaskInput` you create to add it to the `Job`. Each `TaskInput` you add results in one task.
- A `taskCompleted` method. It will be called for each `TaskOutput` that is produced. The method is passed a `taskId` to aid in correlating results with tasks, since they may arrive out of order. `taskId`s are assigned by, and returned from, calls to `addTaskInput`.

JobOptions

Each `Job` is equipped with a `JobOptions` object, which contains various parameter settings. For example, some commonly used options include `PRIORITY` and `GRID_LIBRARY`. A complete list of all options available for the `JobOptions` object is available in the API reference documentation.

Job and Service Comparison

The following table compares Job terms and concepts with the Services model.

	Service	Job
End-to-end work session	Service Session	Job
Atomic unit of work	Service Request Service Response	TaskInput TaskOutput
Implementation	Service Implementation	Tasklet
Collection	Collected immediately. Can also be collected later or never.	Collected after submission. Can also be collected immediately, later, or never.
The synchronization in submit/collect is per Job or Service.		

Summary

- The Tasklet API consists of four types of objects: `Tasklet`, `TaskInput`, `TaskOutput`, and `Job`.
- A Tasklet's `service` method implements the computation that is to be performed in parallel. The `service` method takes a `TaskInput` as argument and returns a `TaskOutput`.
- A `Job` object manages a single `Tasklet` and a set of `TaskInput`s. It is responsible for providing the `TaskInput`s, starting the `Job` and processing the `TaskOutput`s.

Chapter 6
PDriver

PDriver, or the Parametric Job Driver, is a Driver that can execute command-line programs as a parallel processing service using the GridServer environment. This enables you to take a single program, run it on several Engines, and return the results to a central location, without writing a program in Java, C++, or .NET.

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. Scripts could iteratively change the value of variables that are passed to successive tasks as parameters, step through a range of numbers and use each value as a parameter for each task that is created, or define variables containing lists of parameters.

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

Installing PDriver

PDriver is included with the GridServer SDK. For more information on installation, see Chapter 6, "Driver Installation" on page 41 of the *GridServer Installation Guide*.

To use PDriver with SSL, you must configure your Manager to use SSL for all connections. To do this, see Chapter 9, "Configuring Security" on page 72 of the *GridServer Administrator's Guide*.

Resource Deployment

If you plan to run any custom executables using PDriver, they should be deployed to Engines using GridServer's application resource deployment feature, which is described in Chapter 7, "Application Resource Deployment" on page 43 of the *GridServer Administration Guide*. Specifically, resources should be deployed in the `deploy/resources/<platform>/lib` directory, or in another directory that will be referenced by the `execute` command.

PDriver Commands

The following commands are used to run, batch, and cancel PDriver jobs on Unix and Windows systems.

Command	Description
`pdriver`	Starts PDriver, and runs a PDS script.
`bsub`	Starts a one-time Service batch submission.
`bcoll`	Collect a batch jobs on a one-time basis.
`bstatus`	Lists status on pending batch jobs.

Command	Description
bcancel	Cancels a pending batch job.

Before running PDriver for the first time in a shell, you must first set its environment.To set PDriver's environment, either run the setenv.bat or source the setenv.sh file (enter source setenv.sh from the shell prompt), which is located in the pdriver directory of the GridServer SDK.

The pdriver Command

PDriver is started with the pdriver command:

```
pdriver [ -bsub | -bcoll <batchid> | -parallel] [-RA] [-noprompt] [-user driverusername] [-pass
driveruserpassword] [-domain <windowsdomain>] <script>
```

bsub and bcoll stand for batch submission and collection modes. This is useful for submitting long-running Services without tying up the Driver. Submitting a Service in bsub mode creates a Batch Execution with a Batch ID that can be used to run the Service unattended and collect the outputs later. The Batch Executions created are of the same type as those used from the Batch scheduling facility; see Chapter 8, "The Batch Scheduling Facility" on page 61 of the *GridServer Administration Guide* for more information on Batches. This style of batch submission/collection should be used when there are multiple tasks and or prejob/postjob tasks that need to be run and are defined in the PDS script.

RA indicates that the job will be run as the current desktop user, with the current Windows domain. PDriver will prompt for a password. noprompt turns off the password prompt; this is only useful when Service RA auth is disabled on the Broker. domain is Unix only; it allows specification of a win32 domain for Windows Engines to use when authenticating. user and pass are Windows-only, and enable you to pass a Driver user and password (as defined in the GridServer Administration Tool) to PDriver.

The parallel flag is used to run multiple job blocks in a single PDS script in parallel.

The bsub Command

You can also run a one-time Service batch submission with the bsub command:

```
bsub [ -name <name> ] [ -priority <val> ] [ -disc <setting> ] [ -mail <address> ] [ -stdin
<file> ] [ -stdout <file> ] [ -stderr <file> ] [ -nfs ] [-RA] [-domain windowsdomain] [-
noprompt] <app> [args...]
```

This will submit a single Service, defined by <app> [args...], to be scheduled and run on the Grid. The arguments are:

Argument	Description
name	Name of the Service.
priority	Priority of the Service. Allowed values are 1-10.

Argument	Description
disc	Indicates a <name><comparator><value> discriminator expression that must be satisfied for a node to be assigned a Service or task. For example, valid settings include os==win32 or cpuNo>=4. This switch may be used multiple times. Parameters containing a greater-than or less-than symbol must be enclosed in quotes.
mail	Email address to send a confirmation message to when the Service is completed.
stdin	File to be used as input for the Service. This setting is variable based on the -nfs switch, described below.
stdout	File to be used as output when -nfs is enabled.
stderr	File to be used as error output when -nfs is enabled.
nfs	Indicates whether inputs/outputs are available on a shared file system or are to be staged on the Manager. When enabled, -stdin, -stdout, and -stderr function as absolute locators for these files. When not enabled, -stdout and -stderr are ignored, and files designated as <jobname>.out and <jobname>.err are placed in the staging directory. When not enabled, -stdin refers to a file on the Driver file system which is to be staged on the Manager and retrieved by the Engine performing the Service.
RA	Runs the job on Engines as the current desktop user with the current Windows domain. PDriver will first prompt for a password.
domain	Allows specification of a win32 domain for Windows Engines to use when authenticating. This only applies when running PDriver from a Unix machine.
noprompt	noprompt turns off the password prompt; this is only useful when Service RA auth is disabled on the Broker.
<app> [args...]	The application to be run, and any optional arguments. The application is a binary or script that is located on the Engine, and is found in the Engine's path, and args are any arguments passed to the application.

When the Service is submitted, a batch ID is reported to the console. This ID is to be used when collecting outputs (when using the staging directory for input/output rather than NFS mounts.)

The bcoll Command

To collect batch jobs on a one-time basis, use the bcoll command:

```
bcoll <batchid>
```

This is a convenience utility for retrieving the <jobname>.out and <jobname>.err files generated by bsub with nfs mode off. The argument is the batch ID indicated when bsub is finished submitting.

In order to check the status of a Batch Job, you must be running PDriver with a Driver user that has the appropriate access level on the Manager. To do this, create a Driver user with “Manage” level access on the Manager and set via `DSUsername` and `DSPassword` in the `driver.properties` file to this username.

The bstatus Command

To get status on batch jobs, use the `bstatus` command:

```
bstatus [ -jobs ] [ -engines ] [ -stagedir <id> ] [ -canceljob <id> ] [ -batches]
```

`bstatus` accepts the following arguments:

Argument	Description
`jobs`	Lists all jobs on the Manager. This does not include pending batch jobs. The output displays name, jobid, tasks, priority, and status for each job. If there are no jobs, only the four rows of heading lines are returned.
`engines`	Lists all Engines connected to the Manager. The output displays name, ID, os, and status of each Engine. If there are no jobs, only the four rows of heading lines are returned.
`stagedir`	Lists all files living on the staging directory for the given job ID or batchjob ID.
`canceljob`	Cancels a job. This only works for job IDs, not batch IDs, and the Driver must have appropriate permissions for this operation.
`batches`	Lists all Batch jobs on the Manager and their status. The output displays name, batchid, and status of each Batch. If there are no batches, no output is returned.

The `bstatus` command will print its output interspersed with Manager log output on `stderr`. To view only the `bstatus` output on a Unix system, you can redirect `stderr` to null, for example, with `bstatus -jobs 2>/dev/null`. Windows users can use the syntax `bstatus -jobs 2>&0`.

In order to check the status of a Batch Job, you must be running PDriver with a Driver user that has the appropriate access level on the Manager. To do this, create a Driver user with “Manage” level access on the Manager and set via `DSUsername` and `DSPassword` in the `driver.properties` file to this username.

The bcancel Command

To cancel a batch job pending on the Manager, use `bcancel`:

```
bcancel <batchid>
```

The argument is the batch ID returned by `bsub` or `pdriver` in `bsub` mode.

There is an important distinction between batch IDs and Service IDs. A batch ID refers to a Service that is pending execution in the batch queue but has not been handed over to the scheduler. Once that Service is scheduled, a Service with a separate Service ID is launched. The `-canceljob` switch in `bstatus` only works for Service IDs. To remove a pending batch job, `bcancel` must be used. This is unavoidable, since running Services and pending batch jobs are separate entities and are tracked differently. Likewise, to perform output

collection and to see the contents of the staging directory, the batch ID must be used. This is due to the fact that at batch submission, only the batch ID is known, since the Service ID is not generated until run-time. Therefore the batch ID is used as a key for the staging directory.

About PDS Scripts

PDriver uses scripts written in the PDS syntax to define how a Service operates. Aside from defining what programs are run during a Service, PDS scripts enable you to define what happens before and after a task or Service. It also enables you to schedule Services to run in the future, add conditional structure to a Service, and pass custom parameters to a Service or task.

The PDriver script (hereafter referred to as PDS) language allows for the expression of distributed computations that are composed of executable programs. It is designed so that typical computations are easy to describe, while providing for advanced features like conditional execution, iteration, scheduling and discriminators.

This section is a reference for the PDS language. For sample PDriver scripts, see the `examples/pdriver` directory of the GridServer SDK.

PDS Basics

A single PDS file typically corresponds to a single GridServer Service. The computation represented by the Service is usually structured as follows:

1. Split up the input data into several pieces, one per task.
2. Run the tasks in parallel on the Engines.
3. Collect and combine the results.

If the data is too large to be passed as command-line arguments to the program running on the Engine, then it should be placed into files. These files can be located in a shared directory, or they can be copied to the Engine, and the result files copied back.

The PDS language contains constructs for carrying out various statements, such as running executables and copying files, at each point in the lifetime of a Service.

A few words about the lexical structure of PDS: whitespace is not significant, but case is. All text from the "#" character to the end of the line is a comment and is ignored.

PDS Structure

A PDS file begins with the keyword `job` (a synonym for Service) and ends with the keyword `end`. Supply a name after `job` to identify the Service. Two types of elements can occur in between `job` and `end`: parameter declarations, which assign values to variables, and **blocks**, which describe features or statements of the Service. The `options`, `schedule` and `discriminator` blocks describe various facets of the Service, such as when to run it and which Engines can accept its tasks. The other five blocks describe statements to be executed at different phases of the Service. All blocks are optional except for the `task` block. Multiple `job` blocks may also be defined in a single PDS file, and they will be run sequentially by default. You can also run the `pdriver` command with the `-parallel` flag to run multiple jobs in one PDS file in parallel.

The table below details the structure of a PDS file:

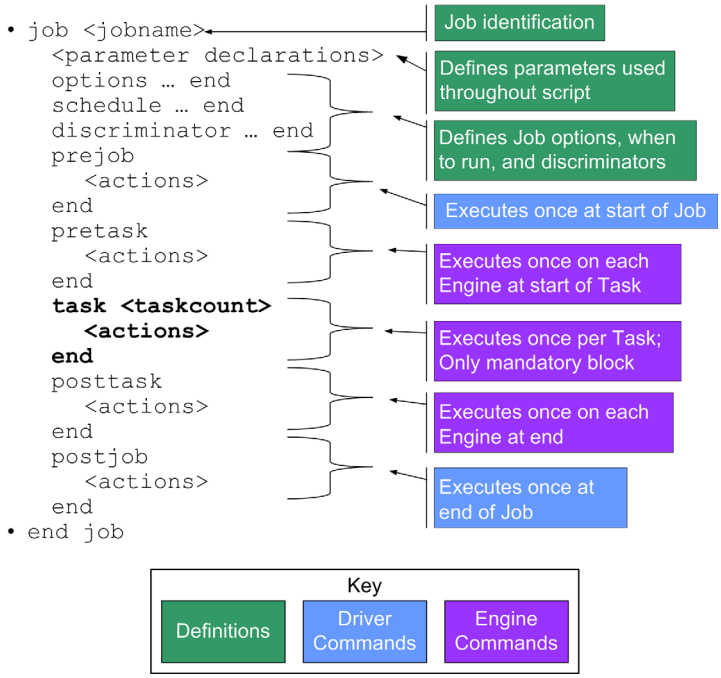


FIGURE 6-1: The structure of a PDS file.

Example 6.1: PDS Structure

```
job <jobname>
options
        onerror ( fail | retry | ignore )
        maxFailedTaskRetries = <val> (default: 3)
        mpiEnabled = <true|false>
        mpiGroupsize = <val> (default: 0)
        enableBlacklisting = <true|false>
        jobPriority = <0-10> (default: 5)
        autoCancelMode = "always" | "never" | "libloadfailure"

        jobOption "<key>" "<val>"
        jobDescription "<key>" "<val>"
end

<variable assignment>

schedule
        <properties>
        [email = "address"]
end

discriminator
        [ affects
                <properties>
          end ]
        <properties>
end
```

Example 6.1: PDS Structure (Continued)

```
prejob
       <statements>
end

pretask
       <statements>
end



task <taskcount>
       <statements>
end

posttask
       <statements>
end

postjob
       <statements>
end

end
```

The Depends Statement

Multiple `job` blocks can be included in a single translation unit (a PDS file and any included PDS files), and by default, they will run sequentially. It is also possible to define jobs that run based on the completion or failure of other jobs using the `depends` statement within a job block. For example, the job containing the following code will run if `firstjob` succeeds, if `secondjob` fails, and following `thirdjob` in either case:

```
depends
       firstjob succeeds
       secondjob fails
       thirdjob succeeds or fails
end
```

The Include Statement

You can include another PDS script within a PDS script by using the `include` statement. For example:

```
include "filename"
```

This includes a PDS script contained within `filename`. The filename can be declared with any string type. The filename is relative to the working directory from which you run PDriver, and can contain a relative or absolute path to a PDS file. The PDS file should contain a full job, and can only be used outside the top-level `job` block. For example, a single PDS file could contain three include statements, each one including a `job` block stored in another file.

Lifecycle Blocks

The five blocks that describe statements to be executed during the life of the Service are discussed below.

prejob

Executes once at the very beginning of the job before any tasks are submitted to the Manager. These commands are executed on the Driver.

pretask

Executes once on every Engine that will be processing tasks for the job, before any tasks are processed. This is intended for generic initialization, such as obtaining input files common to all tasks.

task

Commands in this block are executed once per task. The number of tasks in the Service is determined by the expression following the `task` keyword. Array variables referenced in the task block are indexed by the task ID, as explained in more detail in the Arrays section, below.

Since the same Engine can take more than one task, the statements in the `task` block may run many times on the same Engine. The statements in the `pretask` and `posttask` blocks will run only once per Engine.

The task block is the only required block in a PDS file. Thus a computation that only required executing a program ten times could be expressed by the PDS program:

```
job simple
       task 10
              execute "myprog"
       end
end
```

posttask

Executes once on every Engine at some point after the job has completed. Engine cleanup tasks should go here. Please note however that all files in the `DSWORKDIR` directory will be automatically cleaned by the Engine's file cleaner. The posttask block typically executes after the postjob block executes on the Driver side, although it is not guaranteed to execute then. Execution order of the blocks in a PDS file is typically `prejob`, `pretask`, `task`, `postjob`, and `posttask`, but this is not guaranteed.

postjob

Executes once on the Driver once all tasks have been completed. Typically used for obtaining outputs from the staging directory, running post-processing scripts, and so on.

Other blocks include:

The Options Block

This block is used to set Service options and description information.

Several directives are used for features specific to PDriver jobs:

Keyword	Argument	Description
`onerror`	`ignore`, `retry` or `fail`	Determines the default setting for error-handling. See "Builtin Commands" below for details.
`maxFailedTaskRetries`	A numeric expression	The number of times a failed task will be resubmitted.
`autoCancelMode`	`never`, `always` or `libloadfailure`	Controls whether the entire job is cancelled when a single task fails. `libloadfailure`, which is the default, means that the job will be cancelled if the task failed due to the inability to load a library on the Engine side. Note: this is a standard option and thus can be set with `jobOption`, but the `autoCancelMode` directive supports mnemonic strings as arguments rather than numbers.
`mpiEnabled`	An expression evaluating to `true` or `false`	See "MPI Jobs" below. The default value is `False`.
`mpiGroupSize`	A numeric expression	See "MPI Jobs" below.
`enableBlacklisting`	An expression evaluating to `true` or `false`	This is the same as `engineBlacklisting` for other Services or Jobs. The default value is `false`.

Standard options (which are described in the C++ API reference documentation for the `JobOptions` object) are set with the `jobOption` directive:

```
jobOption "engineBlacklisting" "true"
jobOption "priority" "8"
```

Elements of the job description are similarly set with the `jobDescription` directive:

```
jobDescription "serviceName" "Distributed Grep"
```

Each argument to `jobOption` or `jobDescription` must be a string or an expression that evaluates to a string; literal numbers are not allowed.

The following `jobOption` directives are common to both PDriver and CPPDriver:

Keyword	Argument	Description
`email`	string	An email address which is notified when a Service is completed.
`priority`	integer from 0 - 10	The priority of this Service. The default value is 5.

Keyword	Argument	Description
`taskMaxTime`	integer	If a running task exceeds this amount of time in seconds, the task will be rescheduled or retried, based on the setting of `rescheduleOnTimeout`. The task will be rescheduled or retried when the rescheduler checks for expired tasks after each poll period. This poll period is 60 seconds by default, and can be set on the **Manager Configuration** page, in the **Services** section, under the **Service Rescheduler** heading, with the **Poll Period** property. The default value of `taskMaxTime` is infinite.
`autoCancel`	`0`, `1`, or `2`	Whether the Service is automatically cancelled on a task failure. Possible values include 0 (`AUTO_CANCEL_NEVER`), 1 (`AUTO_CANCEL_LIBRARY_LOAD`), or 2 (`AUTO_CANCEL_ALWAYS`). The default value is 1 (`AUTO_CANCEL_LIBRARY_LOAD`).
`compressData`	`true` or `false`	Whether the tasklet, input, and output data should be compressed. For data sizes greater than10K per input or output, compression time is minimal and is recommended. The default is `false`.
`killCancelledTasks`	`true` or `false`	Whether an Engine will be killed and restarted if a task is cancelled. Tasks are cancelled when cancelled in the Administration Tool, when a Service is cancelled, and when another Engine completes the task due to redundant rescheduling. If this value is `false`, the cancelled method will be called rather than killing the Engine, to provide user-defined interruption of the task and any necessary cleanup. The default value is `true`.
`tasksPerMessage`	integer	The maximum amount of tasks per submission/retrieval message. The messages will not exceed 100 KB, regardless of this number. The default is 100.
`autoPackNum`	integer	The number of tasks per auto-packed tasklet. In this mode, a tasklet will process multiple task inputs in one Service routine by packing task inputs into a single task, and calling your Service routine on all inputs. This mode should be used if there will be many more inputs than Engines, or tasks are of short duration, to maximize efficient use of memory and Engine processing power.

Keyword	Argument	Description
		If inputs are added outside of `createTaskInputs`, the service will check every second if there are any outstanding tasks that have not yet been submitted and submit them in a package even if there are less than the amount requested, to ensure that all tasks are submitted. Task IDs in the Administration Tool will be the IDs of the task packages, so they will not directly correspond to the task ID from the Driver and Engine's point of view. `TaskletException` auto-resubmission and task-level discrimination are not supported for this mode, and will be ignored. Also, this mode is not supported for StreamJobs, and will be ignored if set. The default value is 0.
`autoPackMode`	`0` or `1`	If `0` (`AUTOPACK_SERIAL`), all tasks will be executed serially. If any service method throws an exception that requires an Engine restart, the remaining tasks will fail with a `TaskletException` and not be serviced. If `1` (`AUTOPACK_PARALLEL`), all tasks will be executed in parallel in separate threads. If any Service method throws an exception that requires an Engine restart, all other threads will be interrupted. It is the responsibility of the tasklet implementation to immediately fail by throwing a `TaskletException` if interrupted. The default value is `AUTOPACK_SERIAL`.
`sharedUnixDir`	string	A directory in which the Driver and Unix Engines will exchange data. This directory must be an NFS mounted directory to which all Unix Engines working on this job have read/write access. This will override use of the fileservers on the Driver and Engines, and is optimally a directory local to this Driver for minimum network bandwidth. If set and using Windows Engines, the Windows share directory must also be set to the equivalent of this directory.

Keyword	Argument	Description
`sharedWinDir`	string	A directory in which the Driver and Engines will exchange data. This directory must be a Windows shared directory to which all Windows Engines working on this job have read/write access. This will override use of the fileservers on the Driver and Engines, and is optimally a directory local to this Driver for minimum network bandwidth. Typically, the share will be Windows UNC format, such as `//server/data`. If set and using Unix Engines, the NFS share directory must also be set to the equivalent of this directory.
`checkpoint`	`true` or `false`	Enables checkpointing for this Service. The default value is `false`.
`maxEngines`	integer	The maximum number of Engines that can be working on a task at a time. The default value is infinite.
`statusExpires`	`true` or `false`	Whether the status of the job in the **Service Administration** page expires. If `false`, the status must be manually removed. The default value is `true`.
`engineBlacklisting`	`true` or `false`	Whether Engines that fail at a task should be prevented from taking other tasks from that Service. The default value is `false`.
`unloadNativeLibrary`	`true` or `false`	Whether the native library should be unloaded once the Service is finished. Set the value to `false` for sharing global objects in the library. The default value is `true`.
`deleteInvocationData`	`0`, `1`, `2`, or `3`	How Service invocation data is purged from display in the Administration Tool. This should be set to 0 (`INVOCATION_COMPLETED`) on a Service that will be kept open indefinitely, to avoid memory overhead. The default value is 1 (`INVOCATION_COMPLETED`) for TaskDispatchers, 2 (`SERVICE_COMPLETED`) otherwise. It can also be set to 3 (`MANUAL`) so purging only takes place manually.
`maxTaskRetries`	integer	The maximum number of retries allowed for any task that fails. A retry occurs if the task failed and `serviceFailRetry` is true, or if the task exceed the `taskMaxTime` and `maxTaskReschedules` is false. The default value is 3.
`maxTaskReschedules`	integer	The maximum number of redundant reschedules allowed for any task, if the any of the rescheduler strategies are in effect on the Broker. The default value is 3.

Keyword	Argument	Description
rescheduleOnTimeout	true or false	How a task is dealt with if it exceeds the taskMaxTime. If true, the request is rescheduled, and the current one continues. If false, the Engine running the task is killed, and the task is retried. The default is false.
serviceFailRestart	true or false	Whether an Engine will restart itself on a Service invocation failure. The default is false.
serviceFailRetry	true or false	Whether a Service request will be retried on a failure. If true, it will only be retried up to the maximum numbers of times, as set by maxTaskRetries. The default is false.
allowedDriverProfiles	string	Comma-delimited list of allowed Driver Profiles that may use this Service. All Drivers are allowed if empty.
gridLibrary	string	A Grid Library that is used for this Service. The string argument specifies the name of the Grid Library.
gridLibraryVersion	string	The version of the Grid Library that is used for this Service. The string argument specifies the version of the Grid Library.

JobDescriptions

All Services have a JobDescription object created upon instantiation, with default settings. Predefined properties are stored in the database. Any other properties can be defined. The following JobDescriptions are set by default:

Name	Description
appName	The application name.
appDescription	The application description.
deptName	A department name associated with the Service.
groupName	A group name associated with the Service.
individualName	An individual's name associated with the Service.
serviceName	The name of the Service. By default, this is set to the Service ID.
class	The name of the class in the Service.
serviceType	The type of service: Job, TaskDispatcher

The Discriminator Block

This block specifies either a job-level or task-level discriminator for the job. A discriminator without the `affects` clause is considered a job-level discriminator. Only one job-level discriminator may be declared. Use of the `affects` clause specifies conditions that must be met for the discriminator to be applied to a particular task. Multiple discriminator blocks with `affects` clauses can be used.

Discriminator declarations consist of a property name, a comparator, and a value:

```
<param-name> ==|!=|<|>|<=|>= <param-value>
```

The property name refers to Engine properties. A predefined set of properties are assigned to all Engines by default, and additional properties can be assigned to Engines by the administrator on the **Engine Properties** page and **Engine Property List** page. To see a full list of properties for a given Engine, click the **Engine** tab, click **Engine Admin**, and from the **Actions** list, click **Engine Properties**. Property names are case-sensitive in PDS scripts. For example, the following would discriminate against Engines with a value of less than 350 in the `cpuMHz` Engine property:

```
cpuMHz > 350
```

Use of the affects clause is as follows:

```
discriminator
       affects
              $DSTASKID      #variable
              <              #comparator
              10             #numeric or string
end
```

The variable can be any array type or builtin variable; `$DSTASKID` is the number of the current task, starting with zero. The comparator and numeric or string match against the literal to determine if the discriminator applies against this task. The example above will apply a discriminator to the first ten tasks in a job.

The Schedule Block

Parameters in this block only have an effect if the job is submitted asynchronously with `bsub`. Allowed schedule declarations are:

relative

```
type = relative
minuteDelay = <val>
```

absolute

```
type = absolute
startTime = "mm/dd/yy hh:mm AM|PM"
```

With either type, declaring `email="string"` in the Schedule block will send an email to the address given in the string when the job is complete.

Variables, Types and Expressions

Basics

PDS has two primitive types, string and floating-point number, with the usual notations. Variables need not be declared and can take on values of different types over time. A variable is dereferenced by preceding its name with a dollar sign. Values are converted to the appropriate type depending on context. For instance, a string will be converted to a number when it appears in an arithmetic expression. The grammar forbids certain combinations of expressions to catch common mistakes. Some examples:

```
a = 5.2
b = "2.5"
c = $a + $b          # succeeds, value is 7.7
d = 5.2 + "2.5"      # disallowed by the grammar
```

Scoping

Variables assigned outside of any block are global and visible to all blocks. A variable assigned within a block is visible only within that block.

Inside a block, a variable can be assigned as global variable, which is visible within other related blocks, with the following syntax:

```
global a = 5.2
```

The following table describes what blocks are related with respect to scope:

Global variable set in	Effective in
outside of the lifecycle blocks	all blocks
the `prejob` block	all blocks
the `pretask` block	pretask, task and posttask
the `task` block	task and posttask
the `posttask` and `postjob` blocks	that block

Note that assigning a variable with the same name as a previously defined global variable does not change the value of the global variable. Instead it, creates a new local variable in the block that has local scope and does not change the value of the global variable.

Variable Substitution

Variable references are expanded within quoted strings in all contexts. For example, after

```
a = "foo"
b = "$a fighters"
```

the value of `b` is `"foo fighters"`. Use curly braces to separate a variable name from adjacent non-whitespace text:

```
b = "${a}bar"        # b contains "foobar"
```

Inside of quotation marks, a quote can be represented by escaping it with another quotation mark. Also, inside quotation marks, the dollar sign character can be represented by escaping it with another dollar sign. For example:

```
a = """hello"""      # a contains "hello"
b = "$$100"          # b contains $100
```

The backquote can also be used to assign the output of a shell command to a variable. For example:

```
datetime = `date`
```

Expressions

PDS supports the usual arithmetic operators (`+`, `-`, `*`, `/`) with standard precedence. All arithmetic operations are carried out with double precision floating-point values.

PDS also supports the standard C-style comparison operators (`==`, `!=`, `>`, `<`, `>=`, `<=`). These operators perform numeric comparison if both arguments are valid numbers; otherwise, they perform string comparison. They evaluate to zero if the comparison is false and a non-zero value if it is true.

PDS does not support the relational operators `and`, `or`, and `not`.

Backquote expressions. A string enclosed in backquotes (`` `such as this` ``) has variable substitution performed on it, and the result is evaluated in a subshell. The standard output of the command is collected, newlines and linefeed characters are replaced by spaces, and trailing whitespace is removed. The result is the value of the expression.

Arrays

Arrays are fundamental to achieving parametric parallelism in PDS, because an array variable is implicitly indexed in the task block.

Construction. Arrays of primitive values can be constructed in several ways. A literal value is written thus:

```
a = [1, 2, 3, 4, 5]
```

Arrays can also be constructed by autorange expressions. The expression

```
begin n end m step k
```

constructs an array starting with *n* and proceeding in increments of *k* until *m* is reached. More precisely, it constructs

```
[n, n+k, n+2k, ..., n+rk]
```

where *r* is the largest integer such that $n+rk <= m$.

The expression

```
begin n count c step k
```

constructs an array of *c* elements beginning with *n* and proceeding in increments of *k*, that is,

```
[n, n+k, n+2k, ..., n+(c-1)k].
```

For example,

```
begin 10 end 15 step 2
```

and

```
begin 10 count 3 step 2
```

both construct the array

```
[10, 12, 14].
```

The third way to construct an array is to use the `split` function, which divides a string into array elements at whitespace. Quoted elements will keep embedded whitespace and strip the quotes upon placement into the array. For example, on Unix machines:

```
split(‘ls‘)
```

returns an array of the files in the current directory.

Indexing. In most contexts, when a variable containing an array is referenced, the first element is returned. However, in the task block, the element corresponding to the ID of the currently running task is returned. (If the task ID exceeds the array size, the last element of the array is returned.) This feature makes it easy to write the most common kinds of distributed computations. For example, we can set up an array of values and run a command on each one in parallel with the following PDS script:

```
args = begin 100 end 200 step 5

task sizeof($args)
       execute "doit $args"
end
```

The `$args` in the execute statement expands to 100 for the first task, 105 for the second, and so on.

Another exception to the first element being returned from an array is that inside a `for` loop, the variable containing the array will return the value at the current iteration of the loop. This is redundant with the loop variable. For example, in `for i in $args  log "$i" log "$arg" end`, `$i` and `$arg` will both be the same every time; the loop variable `$i` is only there for convenience.

Explicit array indexing is not supported: the only ways to obtain an array element other than the first are implicit indexing in the task block, and the `for` statement. Even within those contexts, assignment to the variable holding the array changes the entire variable value, not the current element.

Other features. An array that includes both string and numeric values is legal. Arrays of arrays are not allowed.

The number of elements in an array can be determined by the `sizeof` function, as shown in the argument to the `task` block in the above example.

The `for` statement can be used to iterate over the elements of an array. See "The For and Foreach Statement" on page 69, for more information.

Builtin Variables

In addition to user-defined parameters, there are several builtin variables. These are:

Variable	Description
`DSTASKID`	ID of the current task (meaningless if not in task block)
`DSJOBID`	ID of the current job. If submitted using the `bsub` switch, this indicates the batch ID.
`DSWORKDIR`	Engine-side variable indicating temporary directory for job files. This directory is located in the Engine installation directory and is typically `./work/machinename-i/tmp/session-ID`, where machinename is the name of the Engine host; `i` is the instance, starting with 0; and *session-ID* is the Service Session ID for the PDriver Job. Note that the `tmp` directory is periodically deleted by the Engine. The **Temp File Time-to-Live (hours)** setting in each Engine Configuration controls the frequency with which the Engine cleans this directory.
`DSENGINEHOME`	Engine-side variable indicating Engine home
`DSSTAGEDIR`	Alias for the Manager staging directory. This may be used as a source or destination directory for the `copy` command. Files in this directory will be automatically deleted periodically.
`DSOS`	OS of the current system (for example, "linux", "solaris", "win32", "plinux", "zlinux")
`DSMPIGROUPLEADER`	Engine-side variable with value of "true" or "false" indicating whether the current Engine and task are designated as the leader for the current MPI group step. The group leader fetches the routing table from the Manager and executes `mpirun`.
`DSGROUPID`	The current MPI group step. The number of group steps equals the number of tasks divided by the groupsize.

It is important to put any temporary files in the `DSWORKDIR` directory. If you put this elsewhere and a task is interrupted, the files will stay on the computer and never be automatically removed.

Arguments to the PDriver command that follow the script file name may be referenced with the automatic variables `$1, $2, $3 ...` These variables may also be referenced collectively as a string using the automatic variable `$*`. To create an array of all command-line arguments, use `split($*)`.

`$?` contains the execution status of the last `execute` command.

Statements

Builtin Commands

The following are builtin, OS-independent commands that can be used inside any lifecycle block (with the exception of `onerror`, `throw`, and `mpifetch`):

Command	Description
`mkdir "dir-name"`	Creates a directory.
`copy "src-file" "dest-file"`	Copies a file. Typically, the prejob block is used to copy input files from the Driver machine to the staging directory on the Manager, which can be referenced as `$DSSTAGEDIR`. In the task or pretask block, the input file is copied from the staging directory to the Engine's local filesystem. Output files are copied in the reverse direction.
`rmdir "dir-name"`	Removes a directory. The directory must be empty.
`delete "file"`	Deletes a file. The wildcard `*` is supported.
`log "log-message"`	Writes a message to the Driver log (in prejob or postjob lifecycle blocks) or the Engine log (in pretask, task, or posttask lifecycle blocks.)
`execute [stdin="stdin=file"] [stdout="stdout-file"] [stderr="stderr-file"] "command-to-execute"`	Executes a command on the local machine using the shell specified with the `shell` command. A subshell is spawned for the command, unless specified otherwise with the `shell` command. The subshell has the path available on the Engine (as set in the **Library Path** property of the Engine Configuration) or Driver. Before running custom executables, they should be deployed to Engines using GridServer's application resource deployment feature. See "Resource Deployment" on page 49 for more details.
`shell "shell-type"`	Specifies the shell to be used for commands run by the `execute` command. The default is `/bin/bash` for Linux, `/bin/sh` for Solaris, and `cmd.exe` for Windows. This can be set to `none`, whereupon no shell will be spawned.
`env (variable)`	Returns the value of the specified environment variable. If the variable is a string literal, it should be enclosed in quotes. An empty string is returned for nonexistent variables

Command	Description
`onerror <ignore|retry|fail>`	Indicates what will happen when an `execute` command returns with a nonzero exit code. `ignore` will take no action; `retry` will reschedule the task on another Engine; and `fail` will return an exception back to the Driver. The default is `fail`. When used with `retry`, `onerror` can only be used within the `job` or `task` blocks.
`mpifetch <timeout> <filename>`	Fetches a routing table from the Manager containing IP addresses for all Engines in the current MPI group step. The arguments are a fetch timeout and a destination filename for the table. See "MPI Jobs" below for details. `mpifetch` can only be used within the `task` block.
`throw "message"`	Causes the task to fail, and throws an exception. The message is displayed on the Driver and written into the Driver log. This cannot be used in the posttask lifecycle block.

Arguments to all statements except for `onerror` must be enclosed in quotes. Quoted arguments may contain linebreaks only if they are escaped by backslashes.

The `mkdir`, `copy`, `delete`, and `rmdir` commands are not OS-dependent and pathnames will be automatically translated to work on the appropriate platform. For example, `mkdir "sample/log1"` will create `sample\log1` on Windows systems.

The If Statement

The syntax of the basic if statement is

```
if expression comparisonOperator expression then
        statements
end
```

PDS also supports `elsif` and `else` clauses. The `if` statement can be used inside a lifecycle block (`prejob`, `pretask`, `task`, `posttask`, `postjob`), to conditionally execute statements. One typical use is to execute different commands depending on the Engine's operating system:

```
if $DSOS == "win32" then
        cmd = "dir"
else
        cmd = "ls"
end
execute "$cmd"
```

An `if` statement can also appear at the top level of a PDS script, to include or exclude global assignments or entire blocks. An example is

```
if $1 == "alwaysCancel" then
        options
                autoCancelMode "always"
```

```
        end
end
```

The `if` statement is not legal in the `options`, `schedule`, or `discriminator` blocks.

The For and Foreach Statement

Although explicit looping over an array is not often needed, it can be achieved with the `for` statement:

```
for variable in expression
        statements
end
```

The expression must evaluate to an array. The variable takes on successive elements of the array on each iteration of the loop. Assignment to the loop variable is legal, but will have an effect only for the remainder of that iteration. It will not alter the array.

Inside a `for` loop, the variable containing the array will return the value at the current iteration of the loop. This is redundant with the loop variable. For example, in `for i in $args  log "$i" log "$args" end`, `$i` and `$args` will both be equal for each iteration; the loop variable `$i` is only there for convenience.

The `foreach` statement allows you to implicitly index an array, without the loop varaible:

```
foreach expression
        statements
end
```

MPI Jobs

PDriver has support for running MPI Jobs, by using the following two options in the PDS language:

`mpiEnabled` - boolean switch which indicates the job is to be run in MPI mode. An MPI mode job is based on a groupsize (see below), with each group step being treated as a single step of the job. A group step is all groupsize Engines running a step simultaneously. If a single task in an MPI job fails, all other tasks in that group step are rescheduled.

`mpiGroupsize` - The number of nodes used in each MPI group step. The number of tasks for the job must be evenly divisible by this setting.

The above two options go in the options block of a PDS script. Also, the builtin command `mpifetch` can be used in the pretask block to fetch a routing table from the Manager containing IP addresses for all Engines in the current MPI group step. It takes two arguments: a fetch timeout and a destination filename for the table. To determine which node is the MPI group leader, examine the environment variable `DSMPIGROUPLEADER`. It will be set to the string “true” for the Engine chosen as the group leader.

A typical task block for an MPI job would have the following structure:

```
if $DSMPIGROUPLEADER == "true" then
        mpifetch 60 "$DSWORKDIR/route_table"
        # use an execute command to start the MPI computation
else
        # use an execute command to run the MPI daemon
end
```

Note that PDriver MPI jobs may only be run in synchronous mode. Also, if a task fails more than max retries times and an exception is thrown back to the Driver, the job will cancel. There is no way for the Driver to recover in this situation.

MPI support was developed and tested with MPICH 1.2.5. Source code is available for download at http://www-unix.mcs.anl.gov/mpi/mpich/.

See the `examples/pdriver/mpi` directory in the SDK for sample PDS scripts that use MPI.

Shell Directives in Heterogeneous Environments

It is the PDS script writer's responsibility to declare a shell directive that is appropriate to the executing node's platform. For jobs in heterogeneous environments, different directives can be specified within an if-then-else block. For example,

```
if $DSOS == "win32" then shell "cmd.exe" else shell "/bin/bash" end
```

The argument "none" may be specified, in which case all following `execute` tasks are spawned directly and not in a subshell.

It is also possible to use the bash shell with Windows by utilizing Cygwin, the Unix environment for Windows.

To use bash and Cygwin on Windows machines:

1. Install Cygwin on another machine. It can be obtained from `www.cygwin.com`.
2. From the Cygwin installation, copy `bash.exe` and `cygwin1.dll` into the Engine's path. This can be located either in a directory within the `%PATH%` on the Engine, or the replication directory `<engine_home>/resources/win32/bin`. You should also copy any other Cygwin executables you use from the bash shell.
3. If bash is in the path already for the Engine, use the following syntax in the PDS script:

```
task
       shell "bash.exe"
       ...
end
```

or

```
task
       shell "none"
       execute stdout="outfile"
                "<optional_path>/bash.exe -c ""mycomamnd.exe"""
end
```

Or if the path is not set, you can use the following syntax:

```
task
       shell "c:\cygwin\bin\bash"
       execute stdout="outfile" "echo hello"
end
```

The above example also requires you to copy the Cygwin `echo.exe` to the Engine.

PDriver Examples

Included in this distribution are some simple PDriver scripts. These scripts may be found in the `examples/pdriver` directory of the GridServer SDK.

example.pds is a demonstration of the types of commands which can be used with PDriver. The script provides its own contents as a dummy input file. Each Engine prepends a message reporting the current task ID and autorange variable value to this input file. The corresponding outputs are copied back to the Driver at the end of the Service.

pi.pds performs a distributed Monte Carlo calculation of the value of pi. Note that there is a `picalc` commandline executable in the `bin/<platform>` directory of the distribution. It is also available on all Engines in your GridServer installation. Running PDriver will distribute calculation of Pi across the network, using autoranged parameters to generate differing seed values for the program's pseudorandom number generator. The postjob block runs a script which derives an average of all the values returned.

process-dirs.pds demonstrates how the number of tasks may be derived from a comma separated list of values, with each value being processed individually. In this case each branch of a shared directory is given to each Engine for a long directory listing, which is then returned to the Driver. This example only runs on Unix systems with a network-attached file system.

arrays.pds demonstrates usage of the new range types in PDS, as an alternative to using the CSV format.

Chapter 7
GridCache

Introduction

GridCache is a dynamically updateable distributed object cache that enables any GridServer Driver or Engine to store data for later retrieval by other GridServer components. While GridServer makes extensive internal use of object caches, GridCache is an object cache that is explicitly exposed through an API for application code to use directly, to reduce load on backend datastores and to decrease access time to data. GridCache provides a caching system similar to JSR-107 JCache, with an interface as close to JCache as possible and with a subset of features.

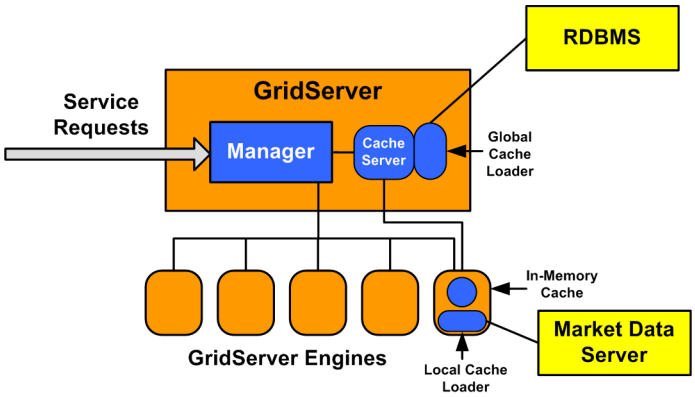


FIGURE 7-1: A typical GridCache implementation.

GridCache is designed to meet 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 every individual state change (for instance every individual quote move) is processed as a transaction by all participants.

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 is stored 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 that has been 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

Access to the GridCache is 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++, with cross-platform access to data provided where appropriate. That is, C++ and Java applications can share XML documents (strings), but would have little use for sharing Java or .NET objects. You can also use Data References across different platforms in order to support streaming very large objects. See “Data References” on page 39 for more information.

Modes

GridCache operates in one of the following modes:

Local – This mode allows a user to cache data locally by putting elements into the cache. There is no synchronization between 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 allows a user to 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 – Users can put data into the cache which 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 load data into the cache using a global loader. Full automatic synchronization occurs in this mode.

Cache Configuration and Access

Cache regions are configured through the **GridCache Configuration** page on the **Services** menu in the GridServer Administration Tool. Region names or regular expressions are defined 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. If a region name matches multiple regular expressions from different schemas, an exception is thrown. A second Cache access method is provided that takes the schema name to provide dynamic mapping of regions to attribute schemas.

Local and Global loaders are configured with the class name of the loader and the type as arguments to the constructor. Users can define bean properties for loaders. Loaders are only available 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 made to the cache configuration will only take effect the next time regions are created with that cache configuration. Pre-existing regions that require the configuration changes have to 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. The Broker's file cache can be size limited, although the default is no limit. If it is size limited and the cache is full, the Broker throws an exception when a user attempts to add new data to the cache rather than silently deleting any data. There is no global resilience when using a memory-based cache; however, a loader can be used for that purpose.

Attributes

Attributes are defined in schemas and applied to newly created regions. The following types of attributed can be defined:

TimeToLive: Regions can define a time to live attribute. Data that has been in the cache for longer than the time to live attribute is evicted, and the user will be forced to reload that data from the backend datastore. For local caches, the data is evicted locally. For distributed regions, the data is evicted from the distributed cache meaning the Broker and all clients delete it if they have cached it locally.

Local/GlobalLoader: Used for loading 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

Cache synchronization is performed by propagating update notifications to all clients listening on a region. These notifications occur any time the region is changed. Specifically, they occur on a put (when an element already existed), 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

A Global Cache Loader is used for synchronized regions from which all clients can accessd 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 is performed and data is not found, 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 `[GS Manager Root]/webapps/livecluster/cache`, or `[Alt Basedir]/cache` if you are using an alternate base directory. The cache directory is configurable in the **Cache** section of the **Manager Configuration** page in the Administration Tool.

Local

A local loader is essentially used to cache data locally from an external database. Local loaders are not shared by clients.

- 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, you invalidate items, which causes that item to be removed from other client's caches.
- Local loaders can only be .NET or Java. CPP loaders can be adapted through JNI or managed C++.

Cache Loader Write-through and Bulk Operations

In order to simplify use of back end datastores in conjunction 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 that such methods can be done in a batch form, such that a cache can be "primed" with entries via a bulk load method or that multiple objects may be stored, removed and invalidated via 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 may be updated, but an invalidation message will fail to be sent to all other clients. If this scenario is detectable, an exception indicating a loss of synchronization will be thrown, but it is up to the cache client to 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 for datastore write-through, bulk write-through, remove, bulk remove and bulk load are available 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

Cache puts (or writes) when the cache is full will push the oldest element out of the cache, possibly into the backing disk cache or possibly removed entirely. This makes the caches LRU caches. Users can configure the size of the local cache and the size of the backing disk cache. For Drivers, configuration is in the `driver.properties` file. For Engines, the configuration is in the Engine configuration. If disk caching is configured, then any puts into the memory cache when the memory cache is full forces 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. There is no disk-backed cache for CPP Drivers.

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. For that reason, it is important to destroy global regions when they are no longer needed, as this will impact eviction performance.

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 will be swept from the cache. A close() method is provided to explicitly release a reference to a region. If the close method is not called, garbage collection will handle 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.

Using The GridCache API

Details about the GridCache API can be found in the GridServer API JavaDocs, available in the GridServer Administration Tool on the **Documentation** page of the **Admin** tab. Documentation for the `Cache` interface covers the use of GridCache.

The GridCache API supports the following primitives:

GridCache constructor with CacheFactory

To create a new GridCache instance, you use the CacheFactory to get a reference to a particular region. On a particular client component, multiple instances of a GridCache can be constructed with the same region, but each exposed instance with the same region shares the same underlying implementation. This allows multiple Sessions to 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, while 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, an attempt is made to load the data for that key from the loader.

Keys

A Key is a string that is used to refer to an object in the cache. The `keys` method gets a list of all keys currently stored on the Manager for this cache for a global region type. For local region types, it gets 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, Manager, and regions on other components.

Invalidation handlers

By default, GridCache implements a lazy invalidation mechanism where the caller is only told that his version of an object is out-of-date when he makes a fresh “get” call for the object. The invalidation handler interface lets the caller register/deregister for asynchronous notification that his local copy of an object has been invalidated by a get, put, remove, or clear.

Fault Tolerance and GridCache

GridCache supports fault-tolerance. For more information, see “GridCache Fault-Tolerance” on page 27 of the *GridServer Administration Guide*.

Chapter 8
GridServer Design Guidelines

This chapter discusses two important aspects to consider when designing an application to run on GridServer, data movement and task or Service request duration. There are a variety of ways to move data among the machines involved in an application; the first section considers their characteristics, and suggests which to choose under various circumstances. In dividing a problem into a set of tasks or Service requests, the programmer can usually choose how many to use, or equivalently, how much time each one should take. The second section discusses factors that can influence this decision.

Data Movement

Every distributed computation ultimately comes down to local computation—a single computing process. Every piece of input data must be moved across the network from wherever it resides to the machine that needs to process it, and every piece of output data must travel over the network from the machine that produced it to its ultimate destination. Additionally, caching can be used to optimize data movement, providing a strategy for lowering the amount of data transfer. Moving large amounts of data over a network efficiently is a crucial aspect in the design of most distributed applications. Efficient data movement can often make a dramatic difference in performance.

Principles of Data Movement

Good data movement design can be summarized in two principles:

Move each piece of data over the network as few times as possible—preferably just once. Obviously, the less that data is moved, the less time it will take to move it. But the many layers of abstraction offered by modern computer systems can hide data movement, making it harder to see the bottlenecks. Network file systems are a good example: there is no way to tell from reading the code whether a file is being read from the local disk or over a network, but the performance difference can be enormous.

Move data as early as possible—preferably before the computation starts. Doing so improves the performance of the computation for the simple reason that the stopwatch that times the computation is started after the data movement has already occurred. But this is more than a mere accounting trick. Consider a nightly report that must be run after 5 PM to avoid conflicting with daytime jobs. If the data for the report is available at 4 PM, it can be distributed to Engines in the hour before the report runs.

Data Movement Mechanisms

Service Request Argument and Return Value The most direct way to transmit data between a Grid client and an Engine is via the argument to a Service request and the return value from that request (task input and task output, in the Job/Tasklet terminology). If Direct Data Transfer is enabled, the data will travel directly between Driver and Engine.

Although each request is handled efficiently, the aggregate data transfer across hundreds of requests can be considerable. Thus any data that is common to all requests should be factored out into session state or init data, or distributed via some other mechanism.

Service Session State Any Service Session (or Job) can have state associated with it. As described in the Services chapters, this state is stored on the Driver as well as on each Engine hosting the instance, so it is fault-tolerant with respect to Engine failure.

Service Session state is ideal for data that is specific to a session. Service Session state is easy to work with, because it fits the standard object-oriented programming model; it is downloaded only once per Engine.

This peer-to-peer data transmission from Driver to Engine is GridServer's Direct Data Transfer feature, enabled by default. When Direct Data Transfer is enabled and a Service creation or Service request is initiated on a Driver, the initialization data or request argument is kept on the Driver and only a URL is sent to the Manager. When an Engine receives the request, it downloads the data directly from the Driver. This mechanism saves one network trip for the data and can result in significant performance improvements when the data is much larger than the URL that points to it, as is usually the case. It also greatly reduces the load on the Manager, improving Manager throughput and robustness.

Shared Directories and DDT In some network configurations, it may be more efficient to use a shared directory for DDT rather than the internal fileservers included in the Drivers and Engines. In this case, the Driver and Engines are configured to read and write requests and results to the same shared network directory, rather than transferring data over HTTP. All Engines and the Driver must have read and write permissions on this directory. Shared directories are configured at the Job and Service level with the `SHARED_UNIX_DIR` and `SHARED_WIN_DIR` options. If using both Windows and Unix Engines and Drivers, you must configure both options to be directories that resolve to the same directory location for the respective operating systems.

Resource Update GridServer's Resource Update mechanism will replicate Grid Libraries, or archives of versioned sets of resources, with Engines. It will also replicate the contents of a directory on the Manager to a corresponding directory on each Engine. Using Resource Update involves using the **Resource Deployment** page in the GridServer Administration Tool to upload files to the Manager. You can also copy the files you want to distribute into a directory on the Manager. Once all currently running Services have finished, the Engines will download the new files. For more on Resource Update, see Chapter 7, "Application Resource Deployment" on page 43 of the *GridServer Administration Guide*

Resource Update is the best way to guarantee that the same file will be on the disk of every Engine in your Grid. File Update is ideal for distributing application code, but it is also a good way to deliver configuration files or static data to Engines before your computation starts. Any kind of data that changes infrequently, like historical data, is a good candidate for distribution in this fashion.

GridCache GridServer's GridCache feature consists of a repository on the Manager that is aggressively cached by components (Drivers and Engines). The repository is organized as a set of regions, each of which is a map from string keys to arbitrary values. The GridCache API supports reads, writes and removing key-value pairs and getting a list of all keys in a catalog. For more information on GridCache, see Chapter 7, "GridCache" on page 73.

A GridCache component caches every value that it gets or puts. If a component changes a key's value or removes it, the Manager asks all components to invalidate their cached copy of that key's value.

GridCache is fault-tolerant with respect to Engine failure, because the data is stored on the Manager. When an Engine fails, its cached data is lost and its task is rescheduled. The Engine that picks up the rescheduled task will gradually build up its cache as it gets data from the Manager.

GridCache is a flexible and efficient way for Engines and Drivers to share data. Like File Update, an Engine needs only a single download to obtain a piece of constant data. Unlike File Update, GridCache supports data that changes over the life of a computation.

GridCache can also be used for Engines to post results. This is generally only useful if those results are to be used as inputs to subsequent computations.

Data References GridServer Data References are objects that represent data existing on a GridServer client. They can be used for passing of lightweight data from one client to another, so that only the destination needing the data performs the data transfer. Typically, the data is stored on the client filesystem, and is served by another client's fileserver.

Data Movement Examples

We illustrate the data movement mechanisms discussed above by variations on a single example: determining the value of a financial instrument. At the heart of this computation is a method we will call value that takes two arguments, a deal and a pricing scenario. The deal argument contains all information specific to a particular financial instrument (bond, derivative security, and so on) needed to determine its value, such as coupon, maturity date, underlying security, and so on. The pricing scenario argument contains all other determinants of the deal's value, such as interest rates, prices of underlying instruments, etc. The output of the value function is a single number representing the value of the deal under the given pricing scenario.

Typical applications require the value of many deals over one or several pricing scenarios. To distribute and parallelize this computation, we execute the value function simultaneously on many Engines. We assume the code for the value function is available to each Engine (whether by Resource Update or over a network file system). We also assume that the numbers returned by the value function make their way back to the client via the standard Service return value mechanism. The question we want to consider is how to get the deal and pricing scenario information to the Engines.

Database Access We first look at the deal information itself, which we assume is stored in a database or data server somewhere on the network. Should the Driver load the deal information from the data server and send it to the Engines as in the left-hand diagram in Figure 8-1, or should the Driver send just the unique identifier and have each Engine access the data server on its own, as illustrated in the right-hand diagram?

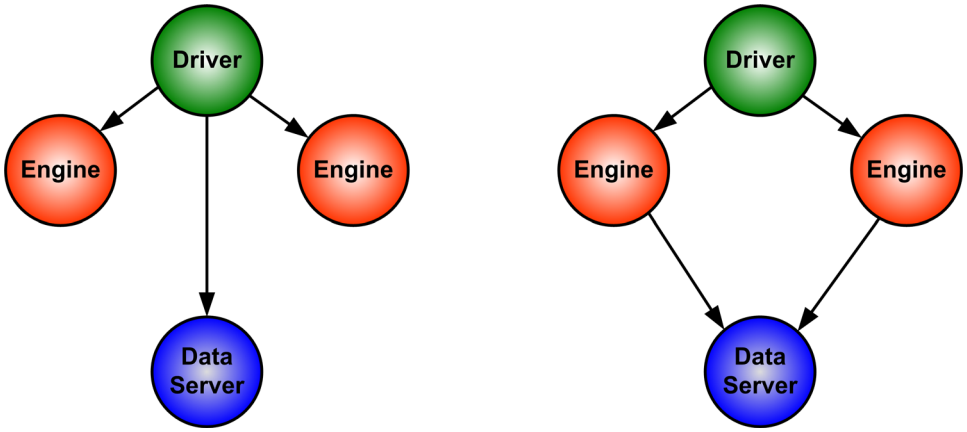


FIGURE 8-1: Data Flow Between A Driver, Two Engines and A Data Server

The second choice is better because less data will move across the network to accomplish the same end result. In the first diagram the data moves across the network twice, once from the data server to the Driver and second from the Driver to the Engine. In the second diagram, the data moves across the network only once from the data server to the Engine. Also, the data only needs to be marshalled and unmarshalled once.

The second approach also increases parallelism at the data server. In the first case, only the Driver is attempting to load data from the data server. In the second, multiple Engines will attempt to load data concurrently. Assuming that the data server can handle the load, the second way will result in increased parallelism.

Single Pricing Scenario We now consider the case in which a single pricing scenario is used to evaluate many deals. Here is one (suboptimal) way to organize this computation. We assume throughout that a Service containing the value function has been deployed and registered.

Algorithm 1 (suboptimal):

1. Create a Service Session of the value service.
2. For each deal, submit the **deal identifier** and the **pricing scenario** as an asynchronous request to the Service Session.
3. Wait for results.

Although this algorithm will get the job done, it needlessly sends the same pricing scenario multiple times. This is an ideal application of Service Session state:

Algorithm 2:

1. Create a Service Session of the value service, initialized with the pricing scenario.
2. For each deal, submit the deal identifier as an asynchronous request to the Service Session.
3. Wait for results.

By making the pricing scenario be part of the session's state, it will be transmitted only as many times as there are Engines that implement the session, rather than once per request. GridServer will never allocate more Engines to a Service session than there are requests for that instance, so Algorithm 2 will never move more data than Algorithm 1. And in the likely event that there are many more requests than Engines (we argue below in the Task Duration section why this is a good idea), Algorithm 2 will move much less data than Algorithm 1.

Several Pricing Scenarios What if the application needs to value the portfolio of deals for more than one pricing scenario? One approach is simply to repeat Algorithm 2 several times, creating a new Service session for each pricing scenario. It is also possible to use a single session and employ the `updateState` method of the Service client API to transmit each successive pricing scenario to the Engines running the session. If the differences between pricing scenarios are small and they are used to perform the update instead of the pricing scenarios themselves, then using `updateState` can result in a considerable data movement savings; even if the pricing scenarios themselves are used as updates, this approach is still likely to be superior to using separate instances.

Multiple Pricing Scenarios Available Early Now let us add the following wrinkle: we still want to compute the value of many deals over many pricing scenarios, but the pricing scenarios are available to us some time before we can run the application. For instance, pricing scenario information may be available at 4 PM, but we cannot start the nightly report until 5:30 PM, to avoid interfering with daily work. In this situation, we can exploit the time gap to push information to the Engines before the computation starts. One approach

would be to use File Update to put all the pricing scenario data on all the Engines. Another would be to put the pricing scenario data into GridCache and run a "primer" Service that copies the data to the Engines. The trade-offs between these two approaches were discussed above under Data Movement Mechanisms.

Deal-Pricing Scenario Symmetry Finally, we point out that deals and pricing scenarios are for the most part symmetric in these examples (the main difference being that pricing scenarios are less likely to be indexed by primary key in a database, so the discussion of deal identifiers versus deal data does not apply to them). For instance, if deals are available to you early, you can use File Update or GridCache to push deal information to Engines before your application starts.

Service or Task Duration

Service or Task duration has an important impact on the performance of distributed computations. Recall that a task corresponds to a single Service request when using Services, or a single task input when using the Job/Tasklet API. Tasks should be long enough to compensate for communication overhead, but not so long that their interruption would seriously delay the overall computation. Dividing the work into more tasks each of which takes less time can also mitigate the performance degradation that can arise from having tasks of different sizes. We discuss these issues in detail in the following sections.

As a running example, we use the deal valuation problem discussed in the previous section on data movement. There we assumed that each task was responsible for pricing a single deal. But this is unlikely to be efficient for most types of deals; instead, several deals should be grouped together in a single task.

Engine Interruption and Smoothing

If an Engine is interrupted or fails during a task, that task will have to run again from the beginning. Therefore, a task should not take a long time to execute. The shorter the task, the less work lost when an Engine fails.

Even if no Engine fails, shorter tasks result in better performance by reducing the variability of task durations. It is rare to know exactly how long every task in a computation will take.

For example, say we divide the work so that there is one task per available Engine, thinking that this minimizes communication overhead and believing that no Engines will fail. However, we estimate task durations wrongly and end up with one task that takes twice as long as the others. Since the computation is not complete until all tasks have finished, the extra-long task will dominate the computation time. For instance, if we have ten Engines, nine tasks that each take one minute, and one task that takes two minutes, then our computation will take two minutes, with the last minute consisting of nine idle Engines and one Engine still working on the two-minute task. With exactly as many tasks as Engines, it is a certainty that our program will run as long as the longest task. (Here and for the rest of the section, we neglect communication time to simplify the discussion.)

If instead we use twice as many tasks as Engines, we significantly improve our expected running time. To continue the above example, if we divide each task in two then we have 20 tasks to give to our ten Engines: 18 tasks of 30 seconds each, and two tasks of one minute each. Each Engine will take two tasks, effectively at random. The chance of the same Engine getting both long tasks is fairly small, so we would expect this program to take one minute 30 seconds most of the time.

Similar reasoning shows that more, shorter tasks smooth out the effect of different processor speeds. Assume that all tasks take the same time, but that one Engine is slower than the others. If there is exactly one task per Engine, the slow Engine will determine the computation time. If there are many, very short tasks, then the slow Engine will take fewer tasks than the other Engines, and all Engines will finish at close to the same time, minimizing the time for the whole computation.

Auto-packing

Because communication overhead involved with each task may make very small tasks inefficient, there is a feature called auto-packing to alleviate this problem. Auto-packing enables you to process multiple requests per task. When set as a Job or Service option, it encapsulates multiple task inputs or Service requests into a single task, which has the overhead of just one task.

To facilitate tuning, it is wise to make task duration, or a related quantity such as number of tasks, a parameter of your application. GridServer's autopacking feature can automate this. Create one task input per item (such as a trade), and set the Job or Service option `AUTO_PACK_NUM` to a positive value to group inputs together. For example, setting `AUTO_PACK_NUM` to 5 will result in each task containing 5 task inputs or Service requests.

Summary

Communication overhead dictates that tasks should take a long time, but the possibility of Engine failure and the opportunity to smooth over differences in task durations and processor speeds suggest that there should be many quick tasks. What is a good compromise? We recommend task running times between 30 seconds and several minutes, and a number of tasks that is three or four times the number of available Engines.

Chapter 9
Using Discriminators

Introduction

In a typical Grid environment, not every machine will be identical. Some machines may be slower, or have less RAM. Other machines may be faster, but it may be a priority to use them to capacity during the day. Depending on the Services you have and the general demographics of your computing environment, the scheduling of Services to Engines may not be clearly deterministic. And sometimes, a specific Service may require special handling to ensure that optimal resources are available for it.

Discrimination is a feature of GridServer that allows you to selectively use Engines based on their properties. Discrimination gives you dynamic control over the connections among Drivers, Brokers and Engines. Discrimination works by specifying which properties an Engine must possess in order to take a Task.

Note: This material is covered in depth with code samples in the *GridServer Object-Oriented Integration Tutorial*. This lesson is designed for readers who want a conceptual reference to discriminators.

Engine Discrimination

Engine Discrimination selects Engines for particular Services based on Engine properties. Engine Discrimination has many uses:

- You can limit a Service to run on Engines whose usernames come from a specified set, to confine the Service to machines under your jurisdiction.
- You can limit a resource-intensive task to run only on Engines whose processors are faster than a certain threshold, or that have more than a specified amount of memory or disk space.
- You can direct a task that requires operating-system-specific resources to Engines that run under that operating system.

You can invent your own properties for Engines and choose them based on those priorities in order to achieve any match of Engines to tasks that you desire.

Setting Discriminators

Discriminators can be set by the Driver, or they can be dynamically attached to a Service based on its Description on the Manager. You can set a discriminator for a Service, or each Task. In Java, the GridServer classes related to discrimination are in the `com.datasynapse.gridserver.discriminator` package. Although there are several classes, you will most likely need to use only `PropertyDiscriminator`.

If you change properties of Engines, these changes will take effect during the execution of a Service. For example, if a discriminator attached to a running Service is configured to look for a certain Engine property, changing this property can change what Engines will work on that Service.

Discriminators are, however, attached to a Service when the Service is created, so changes you make to the discriminator will only affect subsequently submitted Services, not Services that are already running.

For example, to set a Job discriminator in Java:

Example 9.1: Setting a Job Discriminator in Java

```
Properties props = new Properties();
PropertyDiscriminator pd = new PropertyDiscriminator();
// only run on Engines with > 50 Mflops
pd.addComparator(new PropertyComparator(EngineProperties.TOTAL_CPU, "50",
PropertyComparator.GREATER_THAN_EQ, false));
// don't run on win32 Engines
pd.addComparator(new PropertyComparator(EngineProperties.OS, "win32",
PropertyComparator.NOT_EQUALS, false));

job = new SimpleJob(10);
job.getOptions().setDiscriminator(pd);
job.execute();
```

You can also attach discriminators to Services using the **Discriminator Admin** page on the **Services** tab of the GridServer Administration Tool. This page lets you create discriminators by entering two defining factors: the Services that will be affected by the discriminator, and what types of Engines will then be able to run on those Services. This differs from programatic discriminators because they aren't explicitly attached to a Service at its creation; instead, a group of Services is defined as being attached to that discriminator, by Service name, application name, department name, or wildcards on that or other criteria.

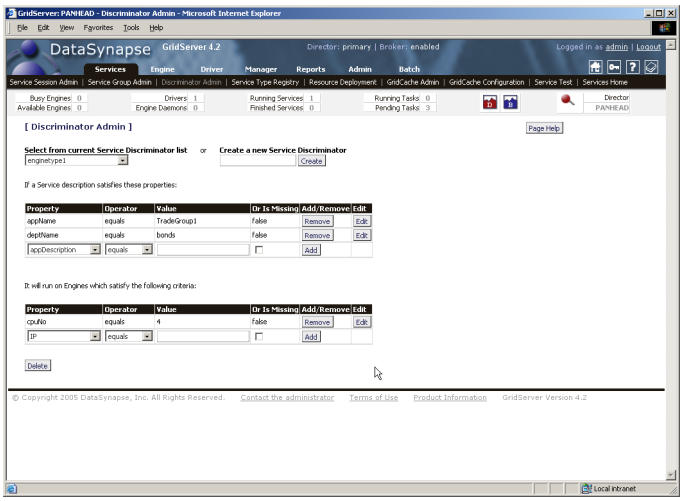


FIGURE 9-1: The **Discriminator Admin** page.

Note that multiple discriminators are ANDed together. For example, if you set two discriminators on a Service, both must be met for an Engine to work on the Service.

Engine Properties

Within GridServer, each Engine has a set of properties. Some Engine properties are set automatically by GridServer, such as the Engine's operating system and the estimated speed of the Engine's processor. Users can also create custom properties for Engines.

Default Properties

An Engine has several properties set by default, with values corresponding to the host machine running the Engine. You can use these properties to set discriminators. The default properties, available in all Engines, can be found in the API documentation, and on the **Discriminator Admin** page, on the **Services** tab.

Custom Properties

In addition to the default properties included in each Engine, you can create your own custom Engine properties, and give them at values Engine installation. There are two steps involved: first, you create the property on the Manager, and then you give it a value for each Engine, either at installation or from the GridServer Administration Tool.

Creating a New Property

To create a new custom property in the GridServer Administration Tool, click the **Engine** tab, then click the **Engine Property List** page. Enter a property name and a brief description, then click **Add**. You can now set a value to this property on any Engine.

Setting a Property Value

To add custom properties to an Engine, in the Administration Tool, click the **Engine** tab, then click the **Engine Properties** page. Select an Engine from the list, and its current properties and values will be displayed. You can select a property name from a list, enter a value, and click **Add**.

You can also simultaneously set properties on a large group of Engines by going to the **Engine Daemon** page of the Administration Tool and selecting **Set Property for Daemons on Page** or **Set Property for All Daemons** from the **Global Actions** list. This opens the **Engine Property Group Editor** page, which displays user-defined properties that can be set on Engines started by a Daemon. Select one or more Engine Daemons from the list, then select a predefined property and enter a value, or enter a new property name and assign a value.

To assign property values on Engine install, you must install the **1-Click Install with Properties** Engine for Windows. You will then be prompted for values for each custom property. If you install a Unix Engine, the default installation script will also ask for values for each custom property on the Manager.

Session Properties

SessionProperties are properties that can be used for discrimination, that last for the duration of an Engine session on the Manager. They are set on an Engine and reset when the Engine logs off.

An example of how to set a Session Property:

```
public void init() {
    // initialize some static data for use by another service
    EngineSession.setProperty("inited", "true");
    // this property can now be used by the discriminator of the other service
}
```

See the API documentation for the `EngineSession` class for more information.

PDriver Discrimination

When writing a PDS script, you can create job-level or task-level discriminators to limit which Engines work on a PDriver job or Task. The `discriminator` block specifies either a job-level or task-level discriminator for a job.

For more information on PDriver discriminators, see “The Discriminator Block” on page 62.

Dependencies

Dependencies allow workflows to be submitted to a Broker without the need of 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 or Tasks already submitted, or ones that have not yet been created. This way, multiple Tasks in different Jobs/Services can be submitted, but they will not be eligible for scheduling until certain conditions are met—namely the successful completion of specific Tasks or Sessions.

Creating Dependencies

Dependencies are exposed as a condition which can be being applied to Services or a Task. This condition has methods for adding dependencies in the form of a Session ID, and for an optional Task ID. Dependencies also allow a Boolean operation that dictates whether a failure of the dependent Task/Service should cause the entire Task/Service to be canceled.

A forward dependency can be created for a Session, and optional Task ID, that does not yet exist. A forward dependency is created by generating a reference ID to that Session, and then using that ID when eventually creating the Session.

In Java, `com.datasynapse.gridserver.client.DependencyFactory` is used to create dependency conditions. More than one dependency can be assigned by using a `ConditionSet`. C++ and .NET APIs are similar. See the GridServer API documentation for more details.

If there is no session with the dependent service ID on the Broker when the session or task with dependencies is added, it is automatically cancelled with the reason that the dependency does not exist. If it is a forward dependency, no such cancellation will be made.

If a Session or Task has a dependency and the dependency failed, the Session or Task can be specified to be cancelled

If a non-forward dependency is made, and the session does not exist, the task is always cancelled.

Administering Task Dependencies

Dependencies can be viewed in the GridServer Administration Tool on the **Service Session Admin** page on the **Service** tab, or from the **Task Admin** page, available from the **Actions** list on the **Service Session Admin** page. and Service Admin pages. On either page, select **Service Session Details** or **Task Details** from the **Actions** list, and in the details, a list of dependencies will be shown, noting which are pending and which have completed.

Dependencies can be removed from a Task or Service with the **Remove Dependencies** action on the **Actions** list on the **Task Admin** page. This will remove the entire Dependancy object, which removes all pending dependencies; there is not a way to remove a single dependency from the Administration Tool.

Note that Task Dependencies are Broker-scope, and rely on Service and Task events on a Broker. They do not work across Brokers.

Chapter 10
GridServer Admin API

Introduction

The GridServer Admin API offers programmatic access to administrative tasks and information normally performed or presented in the web-based GridServer Administration Tool. It is available via Java, C++, and .NET Drivers, on Managers via Manager Hook, and from SOAP Web Services.

Documentation for the GridServer Admin API

Detailed documentation on GridServer Admin API can be found in the API documentation. Also, the WSDL for Services can be retrieved at the **Web Service List** page on the **Manager** tab in the GridServer Administration Tool.

The following components are defined:

- BatchAdmin
- BrokerAdmin
- DriverAdmin
- EngineAdmin
- EngineDaemonAdmin
- ManagerAdmin
- ServiceAdmin
- DriverManager

All GridServer API methods require HTTP basic authentication, and the methods allowed are based on the access level of the user. Note that methods will return null if there is no output, as opposed to returning a zero-length array. For example, `EngineAdmin.getAllEngineInfo()` will return null if there are no Engines currently logged into the Broker.

Drivers services require authentication and an associated Driver Profile with the user if Driver Authentication is enabled.

Access Level Requirements and Availability for Admin API

Admin API functionality is limited according to a user's Access Level, which is a tiered security level assigned to each GridServer user's account. There are four Access Levels: View, Service, Manage, and Configure. Also, some Admin API functionality varies, depending on the components available in a Manager. The following table shows what methods in each class are available for each Access Level:

Access Level	Class	Method
View level		
	ServiceAdmin	isAvailable, getServiceCount, getRunningServiceCount, getCompletedServiceInvocationCount, getFinishedServiceCount, getServiceInvocationCount, getRunningInvocationCount, getPendingInvocationCount, getRunningServiceInvocationCount, getPendingServiceInvocationCount, getServiceInfo, getAllServiceInfo, getInvocationCount, getInvocationInfo, getSelectedInvocationInfo, getSelectedServiceInfo, getServiceIds
	EngineAdmin	isAvailable, getEngineInfo, getAllEngineInfo, getEngineCount, getBusyEngineCount, getEngineIds, getSelectedEngineInfo.
	EngineDaemonAdmin	isAvailable, getEngineDaemonCount, getEngineDaemonInfo, getAllEngineDaemonInfogetEngineDaemonIds, getSelectedEngineDaemonInfo
	DriverAdmin	getDriverCount, getDriverInfo, getAllDriverInfo, getDriverProfileCount, getDriverProfile, isAvailable, getAllDriverProfiles
	BrokerAdmin	isAvailable, getBrokerCount, getAllBrokerInfo, getBrokerInfo
	ManagerAdmin	isAvailable, getBrokerUrl, getLicenseInfo, getVersion, getBusyEngineCount, getEngineCount, getFinishedServiceCount, getPendingInvocationCount, getRunningInvocationCount, getRunningServiceCount, getServiceCount
	BatchAdmin	isAvailable
Service level		
Contains all from the View Level plus the following:		
	BrokerAdmin	getDescriptionDiscriminators
	EngineAdmin	getLogUrlList,
	EngineDaemonAdmin	getLogUrlList
	ServiceAdmin	cancelService, cancelAllServices, removeFinishedService, removeAllFinishedServices, setPriority, cancelInvocation, getServiceBinding, getRegisteredServices, setExpires
Manage level		
Contains all from the Developer Level plus the following:		
	EngineAdmin	killEngine, killAllEngines
	EngineDaemonAdmin	setProperty, removeProperty restartEngineDaemon, setAllEnabled, setAllStartMode, setConfiguration, setEnabled, setInstances, setStartMode, restartEngineDaemonByComparator, setConfigurationByComparator, setEnabledByComparator, setInstancesByComparator, setStartModeByComparator
	BrokerAdmin	addDescriptionDiscriminator, deleteDescriptionDiscriminator, addDriverRoutingComparator, addEngineRoutingComparator removeDriverRoutingComparator, removeEngineRoutingComparator, setDriverWeight, setEngineWeight, setMaximumEngines, setMinimumEngines
	Manager Admin	getEvents, getSubscribers, getSubscriberEvents

Access Level	Class	Method
	BatchAdmin	getAllBatchInfo, getBatchCount, getBatchInfo, getRunningBatchCount, removeBatch removeFinishedBatches, resumeBatch, suspendAllBatches, suspendBatch, getBatchExecutionCount, getBatchExecutionIds, getBatchExecutionInfo, getBatchExecutionInfoByBatchId, getBatchIds, getRunningBatchExecutionCount, getScheduledBatchCount, getSelectedBatchExecutionInfo, getSelectedBatchInfo, removeBatchExecution, removeFinishedBatchExecutions, getAllBatchExecutionInfo

Configure level

Contains all from the Manage Level plus the following:

ServiceAdmin	registerService, unregisterService
EngineDaemonAdmin	getDefaultProperties, setDefaultProperty, removeDefaultProperty, setDirectors, setDefaultCodeVersion, setDefaultCodeVersionByComparator, setDirectorsByComparator
DriverAdmin	addDriverProfile, deleteDriverProfile, setDefaultProperty, removeDefaultProperty, getDefaultProperties
ManagerAdmin	subscribe, unsubscribe, getCategoryNames, getCategory, setValue
BatchAdmin	getBatchFileNames, addBatchFile, deleteBatchFile, getBatchFile, scheduleBatchFile, addBatchDefinition, deleteBatchDefinition, getBatchDefinition, getBatchDefinitionNames, scheduleBatchDefinition

The following table shows what Manager components are required to use Admin API classes and methods:

	Class	Method
Requires Broker:	ServiceAdmin	All
	EngineAdmin	All
	DriverAdmin	All
	BatchAdmin	All
Requires Director:	EngineDaemonAdmin	All
	BrokerAdmin	All
	ManagerAdmin	getEngineCount, getBusyEngineCount, getServiceCount, getRunningServiceCount, getFinishedServiceCount, getPendingInvocationCount, getRunningInvocationCount
Requires Primary Director:	ServiceAdmin	registerService, unregisterService
	EngineDaemonAdmin	removeProperty, setProperty, setDefaultProperty, removeDefaultProperty

	Class	Method
Requires Primary Director: (cont.)	DriverAdmin	deleteDriverProfile, addDriverProfile
	BrokerAdmin	All except getBrokerCount, getBrokerInfo, getAllBrokerInfo

Using The ServiceClient Web Service

Services can be created and run by using the ServiceClient Web Service. The following explains how to use the ServiceClient Web Service:

- Use the DriverManager service's `getWebServicesURL` on a Director to retrieve the web service's URL of an appropriate Broker. This service should then be used. Use the `ping` method to keep the session alive in the absence of any other activity.
- The `createService`, `execute`, `destroy` and `updateState` methods behave as they do in the language APIs (see Chapter 3, "Creating Services" on page 23 for details and examples).
- Asynchronous submission is accomplished via the `submit`, `collect`, and `collectAck` methods. Each call to submit returns an integer request ID unique to that call. Poll for results periodically by invoking `collect`, which returns the result of every completed request, matched with their request IDs. After `collect` returns, invoke `collectAck` to acknowledge receipt of the request results, so that completed requests can be purged from the Manager.

Using the Admin API over SOAP

The following is an example of using the Admin API over SOAP with Java:

1. Locate the WSDL for the Admin Service from the Manager's Web Service List. For example, to use the `EngineDamonAdmin` class, `http://example:8000/livecluster/webservices/EngineDaemonAdmin?wsdl`
2. Generate Java Stubs for the Service. For example, using Axis:

```
org.apache.axis.wsdl.WSDL2Java
http://example:8000/livecluster/webservices/EngineDaemonAdmin?wsdl
```

3. Use the Stubs. For example:

```
// Get the interface to the Admin Service
EngineDaemonAdmin server = (new EngineDaemonAdminServiceLocator()).getEngineDaemonAdmin();

// Required when Driver authentication is enabled
((Stub)server).setUsername("admin");
((Stub)server).setPassword("admin");

// Maintain the session id for each request
((Stub)server).setMaintainSession(true);

// Query the Admin Service
EngineDaemonInfo[] info = server.getAllEngineDaemonInfo();
```

Chapter 11
Extending GridServer

Introduction

The GridServer Manager and Engine can both be extended with Manager and Engine Hooks. A Manager Hook enables you to interface your own Java object directly with the Manager's event processing mechanism, and interact with any Server Event. An Engine Hook can perform user-defined operations on Engine startup or termination.

A Hook consists of two parts: the class implementation of the Hook, and the Hook registration. Manager Hooks are registered on the **Hook Admin** page, while Engine Hooks are registered with an XML file that is deployed to Engines.

Manager Hooks

Manager Hook implementation details are discussed in the JavaDoc documentation for the `ServerHook` class. You will also find examples on creating a Manager Hook in the GridServer SDK, in the `examples/hooks/server` directory. After implementing a Manager Hook, its class definition must be contained in a JAR file in the shared classes directory (`WEB-INF/hooks/`*`component`*, where *`component`* is either `broker` or `director`).

Manager Hooks can be created on the Broker or the Director. Depending on where the hook resides, it can receive a different subset of the Server Events broadcast by the Manager. See the JavaDoc for the `ServerEvents` class for more details on the events available and which components to which they are relevant. For example, a hook that needs to be aware of when an Engine has been added or removed would have to run on the Broker, because the `ENGINE_ADDED` and `ENGINE_REMOVED` events are Broker-specific.

To register a Manager Hook in the GridServer Administration Tool, click the **Admin** tab and click **Hook Admin**. The **Hook Admin** page enables you to edit, enable, or disable hooks on the Manager To add a new hook, select **Create New Hook** from the **Global Actions** list. This opens a Hook Editor in a new window. Enter a filename for the hook XML file, and select if the hook will be applied to the Director or Broker. Enter the name of a class in the hooks directory and click **Update Properties** to display an updateable property list below. After you have entered any properties, click **Save** to edit the hook, or **Cancel** to revert to the last saved version of the hook.

From the **Actions** control of each hook, you can Enable, Disable, Edit, or Delete an existing hook. Note that the Manager does not need to be restarted after deploying the JAR. However, if you redeploy a JAR, you must remove and re-add the hook for any new changes to take effect.

Engine Hooks

Engine Hook implementation details are discussed in the JavaDoc documentation for the `EngineHook` class. You will also find examples on creating an Engine Hook in the GridServer SDK, in the `examples/hooks/engine` directory. The following is an example that initializes a JDBC database.

Example 11.1: JDBCHook.java

```
// Copyright 2002 DataSynapse, Inc. All Rights Reserved
package examples.hook;

import com.datasynapse.gridserver.engine.*;
import java.sql.*;
import java.util.*;

/*
 * This is an example of a hook that initializes data from a database.
 * The property "initialized" can be used to discriminate on jobs, so that
 * only engines that have initialized the data will take tasks.
 */

public class JDBCHook extends EngineHook {
    public void initialized() {
        try {
            initializeData();
            EngineSession.setProperty("initialized", "true");
        } catch (Throwable e) {
            System.err.println(e);
        }
    }

    // static method is used by the tasklet
    public static Vector getData() {
        return vData;
    }

    private void initializeData() throws ClassNotFoundException {
        System.out.println("initializing");
        Class cl = Class.forName(getDriver());
        System.out.println("Driver class:" + cl);

        boolean successful = false;
        do {
              try {
                Connection conn = DriverManager.getConnection(getUrl(),
                                        getUsername(), getPassword());
            PreparedStatement ps = conn.prepareStatement("select * from people");
                ResultSet rs = ps.executeQuery();
                System.out.println("rs:" + rs);
                while ( rs.next() )
                    vData.add( rowToLine(rs) );
```

Example 11.1: JDBCHook.java (Continued)

```
    successful = true;
                } catch (Exception e) {
                    System.out.println("JDBCHook: failed to retrieve data, will try
    again.");
                }
                if (!successful) {
                  try { Thread.sleep(getFrequency()); } catch (InterruptedException
    ie) { break; }
                }
    } while (!successful);
        }

    static String rowToLine( ResultSet input ) throws SQLException {
            StringBuffer buf = new StringBuffer();
            int cols = input.getMetaData().getColumnCount();
            for ( int i=1; i <= cols; i++ ) {
                buf.append(input.getString(i));
                buf.append(' ');
            }
     buf.append('\n');
            return buf.toString();
        }


        public final void setUrl(String url) {
            _url = url;
        }

        public final String getUrl() {
            return _url;
        }

        public final String getDriver() {
            return _driver;
        }

    public final void setDriver(String driver) {
            _driver = driver;
        }

        public final String getUsername() {
            return _user;
        }

        public final void setUsername(String user) {
            _user = user;
        }

        public final String getPassword() {
            return _pass;
        }
```

Example 11.1: JDBCHook.java (Continued)

```
    public final void setPassword(String password) {
            _pass = password;
        }

        public final void setFrequency(long frequency) {
            _frequency = frequency;
        }

        public final long getFrequency() {
            return _frequency;
        }

        private String _url;
        private String _driver;
        private String _user;
        private String _pass;
        private long _frequency = 5000;

        private static Vector vData = new Vector();
    }
```

The class definition for the Hook must be contained in a JAR file in the shared classes directory (either in `deploy/resources/shared/jar` on the Manager or the shared NFS mount as defined in the Engine configuration) or in a Grid Library.

Hooks are added by adding the XML file to the `deploy/resources/shared/hook` directory, or adding them to a Grid Library. You can add several different XML files to this directory (as opposed to the method of having a single `hooks.xml` file, used in previous releases.) An example of the XML format to use for your Hook is documented in the `EngineHook` JavaDoc. The following is also an example of the XML to add to the `hooks.xml` file for the JDBC example given above.

Example 11.2: XML for JDBCHook

```
<hook class="examples.hook.JDBCHook">
        <property name="username" value="sa"/>
        <property name="password" value=""/>
        <property name="url" value="jdbc:HypersonicSQL:hsql://%server%:2034"/>
        <property name="driver" value="org.hsql.jdbcDriver"/>
</hook>
```

Chapter 12
API Extensions

Introduction

This chapter covers classes that are extensions to the GridServer Tasklet API. The GridServer API documentation covers this information in more detail.

Note: This material is covered in depth with code samples in the *GridServer Object-Oriented Integration Tutorial*. This chapter is designed for readers who want a conceptual reference to the API extensions.

StreamJob and StreamTasklet

The `service` method of a standard GridServer tasklet uses objects for both input and output. These `TaskInput` and `TaskOutput` objects are serialized and transmitted over the network from the Driver to the Engines. For some applications, it may be more efficient to use streams instead of objects for input and output. For example, applications involving large amounts of data that can process the data stream as it is being read may benefit from using streams instead of objects. Streams increase concurrency by allowing the receiving machine to process data while the sending machine is still transmitting. They also avoid the memory overhead of deserializing a large object.

A `StreamJob` is a Job which allows you to create input and read output via streams rather than using defined objects. A `StreamTasklet` reads data from an `InputStream` and writes to an `OutputStream`, instead of using a `TaskInput` and `TaskOutput`. `StreamJob` writes input to a stream, and the Tasklet code on the Engine reads data from this stream. In this way, the memory overhead on the Driver, Broker, and Engine is reduced, since an entire `TaskInput` does not need to be loaded into memory for transfer or processing. The `StreamTasklet` must be used with a `StreamJob`.

Use `StreamTasklet` and `StreamJob` when the amount of input or output data is large, and a tasklet can process the data stream as it arrives. The `service` method of `StreamTasklet` reads its input from an `InputStream` and writes its results to an `OutputStream`. When writing a `StreamJob` class, create an input for a task by calling the `createTaskInput` method to obtain an `OutputStream`, then writing to and closing that stream.

The `TaskCompleted` method of `StreamJob` is given an `InputStream` to read a task's results. It is your responsibility to close all streams given to you by GridServer.

DataSetJob and TaskDataSet

A data set is a persistent collection of task inputs (either `TaskInput` objects or streams) that can be used across jobs using different Tasklets. The first time it is used, the data set distributes its inputs to Engines in the usual way. But when the data set is used subsequently, it attempts to give a task to an Engine that already has the input for that task stored locally. If all such Engines are unavailable, the task is given to some other available Engine, and the input is retransmitted. Data sets thus provide an important data movement optimization without interfering with GridServer's ability to work with dynamically changing resources.

A `TaskDataSet` is a collection of TaskInputs that persist on the Manager as the input for any subsequent `DataSetJob`. The TaskInputs get cached on the Engine for subsequent use for the `TaskDataSet`. This API is therefore appropriate for doing repeated calculations or queries on large datasets. All Jobs using the same `DataSetJob` will all use the `TaskInputs` added to the `TaskDataSet`, even though their Tasklets may differ.

Also, `TaskInputs` from a set are cached on Engines. An Engine that requests a task from a Job will first be asked to use input that already exists in its cache. If it has no input in its cache, or if other Engines have already taken input in its cache, it will download a new input, and cache it.

An ideal use of `TaskDataSet` would be when running many Jobs on a very large dataset. Normally, you would create `TaskInputs` with a new copy of the large dataset for each Job, and then send these large `TaskInputs` to Engines and incur a large amount of transfer overhead each time another Job is run. Instead, the `TaskDataSet` can be created once, like a database of `TaskInputs`. Then, small Tasklets can be created that use the `TaskDataSet` for input, like a query on a database. As more jobs are run on this session, the inputs become cached among more Engines, increasing performance.

To create a `TaskDataSet`, first construct a new `TaskDataSet`, then add inputs to it using the `addTaskInput` method. If you are using a stream, you can also use the `createTaskInput` method. After you have finished adding inputs, call the `doneSubmitting` method. If a name is assigned using `setName`, that will be used for subsequent references to the session; otherwise, a name will be assigned by GridServer. The set will remain on the Manager until `destroy` is called, even if the Java VM that created it exits.

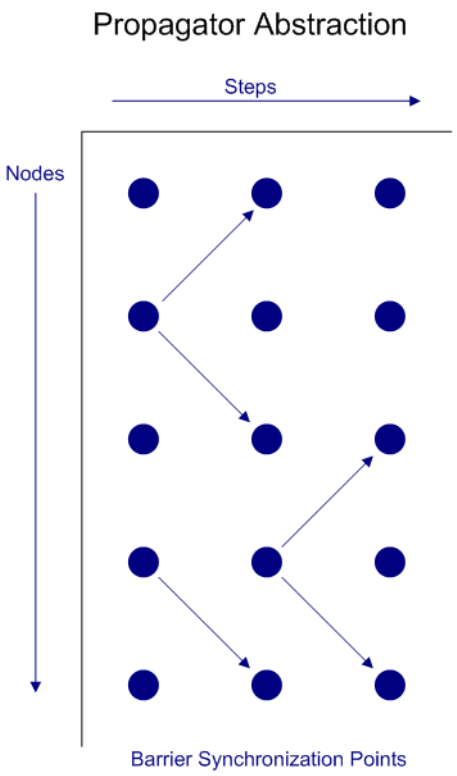
After creating a `TaskDataSet`, implement your Job using `DataSetJob`. The main difference is that to run the Job, you must use `setTaskDataSet` to specify the dataset you created earlier. Note that the `executeLocally` test method cannot be used with the `DataSetJob`.

The Propagator API

The Propagator API is an appropriate alternative to MPI for running parallel computations which require inter-node communication. Unlike most MPI implementations, Propagator implementations can run over heterogeneous resources, including interruptible desktop PCs.

A Propagator application is divided into steps, with steps sent to nodes. Using adaptive scheduling, the number of nodes can vary, even changing during a problem's computation. After a step has completed, a node can communicate with other nodes, propagating results and collecting information from nodes that have completed earlier steps. This checkpointing allows for fault-tolerant computations.

The following diagram illustrates how nodes communicate at barrier synchronization points when each step of an algorithm is completed:



Using the Propagator API

The Propagator API consists of two classes, `GroupPropagator` and `NodeTasklet`, and the Interface `GroupCommunicator`.

- The `GroupPropagator` is used as the controller. It is created and used to create the nodes and the messaging system used between nodes.
- The `NodeTasklet` contains the actual code that each node will execute at each step. It also contains whatever code each node will need to send and receive messages, and send and receive the node state.
- The `GroupCommunicator` is the interface used by the nodes to send and receive messages, and to get and set node state.

GroupPropagator

The `GroupPropagator` is the controlling class of the `NodeTasklet`s and `GroupCommunicator`. You should initially create a `GroupPropagator` as the first step in running a Propagator Job.

You can create a `GroupPropagator` and access the `GroupCommunicator`, like this:

```
GroupPropagator gp = new GroupPropagator("simple", nodes);
GroupCommunicator gc = gp.getGroupCommunicator();
```

This will enable you to communicate with nodes, and get or set their state.

Next, you will need to set the `NodeTasklet` used by the nodes. Given a simple `NodeTasklet` implementation called `TestPropagator` that is passed the value of the integer `x`, you would do this:

```
gp.setNodeTasklet( new TestPropagator( x ) );
```

After you have defined a `NodeTasklet`, you can tell the nodes to execute a step of code by calling the `propagate` method, and passing a single integer containing the step number you wish to run.

When a program is complete, the `endSession` method should be called to complete the session.

NodeTasklet

The `NodeTasklet` contains the actual code run on each node. The `NodeTasklet` code is run on each step, and it communicates with the `GroupCommunicator` to send and receive messages, and set its state.

To create your own `NodeTasklet` implementation, create a class that extends `NodeTasklet`. The one method your class must implement is `propagate`. It will be run when `propagate` is run in the `GroupPropagator`, and it contains the code your node actually runs.

The code in the `NodeTasklet` will vary depending on the problem. But several possibilities include getting the state of a node to populate variables with partial solutions, broadcasting a partial solution so that other nodes can use it, or sending messages to other nodes to relay work status or other information. All of this is done using the `GroupCommunicator`.

GroupCommunicator

The `GroupCommunicator` communicates messages and states between nodes and the `GroupPropagator`. It can also transfer the states of nodes. It's like the bus or conduit between all of the nodes.

The `GroupCommunicator` exists after you create the `GroupPropagator`. It's passed to each `NodeTasklet` through the `propagate` method. Several methods enable communication. These methods are described in more detail in the JavaDoc API documentation. This list includes methods commonly used; there are variations available to delay methods until a specified step or to execute them immediately.

Method	Description
`broadcast`	Send a message to all recipients, except current node.
`clearMessages`	Clear all messages and states on Manager and Engines.
`getMessages`	Get the messages for current node.
`getMessagesFromSender`	Get the message from specified node for current node.
`getNodeState`	Get the state of specified node.
`getNumNodes`	Get the total number of nodes.
`sendMessage`	Send the message to nodeId.
`setNodeState`	Set the state of the node.

A Propagator API Example

The remainder of this chapter will explain how to use the Propagator API to solve a simple one-dimensional heat equation. The state of each node will consist of a single number. On each step, each node will adjust its state in proportion the difference between its current value and the value of its neighbors. The Java source is included with the SDK in the project "`simplepropagator`", so you can build, test, and modify the code. This example uses three files: `Test.java`, which contains the main class, `SimplePropagator.java`, which implements the `NodeTasklet`, and `Result.java`, which stores the results from each node.

Test.java This file starts like most other GridServer programs, except we import `com.livecluster.tasklet.propagator.*`. Also, a Test class is created as our main class.

Example 12.1: Test Main Class

```
package examples.simplepropagator;

import com.livecluster.tasklet.propagator.*;
import com.livecluster.tasklet.util.*;
import java.lang.*;
import java.util.*;
import java.io.*;
import java.text.DecimalFormat;

public class Test  {
        public static void main(String[] args) throws Exception {
```

The main method begins by creating a `GroupPropagator` with the name "simple" and a hard-coded number of nodes. A new `SimplePropagator` is created and set into the `GroupPropagator`, which is the code for the `NodeTasklet`. `SimplePropagator` is described in the next section. Then, a `GroupCommunicator gc` is assigned with the `GroupPropagator` method `getGroupCommunicator`.

Example 12.2: GroupCommunicator Assignment

```
        int nodes = 6;
        int steps = 12;
        GroupPropagator gp = new GroupPropagator("simple", nodes);
        try {
            gp.setNodeTasklet(new SimplePropagator(steps, 0.5));
            GroupCommunicator gc = gp.getGroupCommunicator();
```

The boundary conditions are set by initializing the states of the leftmost and rightmost nodes. Then, for each step, the `GroupPropagator`'s `propagate` method is called and the results are printed. We finish by ending the session.

Example 12.3: Main Loop

```
            // Set initial state
            gc.setNodeState(0, 0, new Double(10));
            gc.setNodeState(nodes-1, 0, new Double(8));

            for (int i = 0; i < steps; i++) {
                Object[] results = gp.propagate(i);
                printResults(i, results);
            }
        } finally {
            gp.endSession();
        }
    }
```

The `printResults` method is given the return value of `propagate`, which consists of an array of `Result` objects. Since these may be returned in any order, they are sorted by node ID. Then they are formatted and displayed.

Example 12.4: printResults

```
    static void printResults(int step, Object[] results) {
        Arrays.sort(results);
        DecimalFormat fmt = new DecimalFormat("###0.00  ");
        System.out.print(step + "\t");

        for (int i = 0; i < results.length; i++) {
            double d = ((Result) results[i]).value;
            System.out.print(fmt.format(d));
        }
        System.out.println();
    }
}
```

TestPropagator.java. The class `TestPropagator` is defined with a constructor that is passed the number of steps and a value used to calculate the equation:

Example 12.5: TestPropagator Class

```
package examples.simplepropagator;

import com.livecluster.tasklet.propagator.*;

class SimplePropagator extends NodeTasklet {
    SimplePropagator(int steps, double fac) {
        _steps = steps;
        _factor = fac;
    }
```

`SimplePropagator`'s propagate method is where all the work is done. It uses the `GroupCommunicator` to get its state for this step and the messages its neighbors sent to it from the previous step. It uses this information to calculate its next state.

Example 12.6: propagate and printResults Methods

```
    public Object propagate(int nodeId, int stepId, GroupCommunicator gc)
                            throws Exception {
        // Get our own state from the previous step.
        double current = toDouble(gc.getNodeState());
        double next = current;
        //  Get neighbors' messages containing their previous values.

            if (nodeId != gc.getNumNodes() - 1) {
                double right = toDouble(gc.getMessagesFromSender(nodeId+1)[0]);
                next += _factor*(right - current);
            }
        }

    static void printResults(int step, Object[] results) {
        Arrays.sort(results);
        DecimalFormat fmt = new DecimalFormat("###0.00  ");
        System.out.print(step + "\t");
```

The node then sets its state for the next step and sends its state to its immediate neighbors. Finally, it returns its state along with its node ID in a `Result` object:

Example 12.7: Inform Neighboring Nodes, Set State

```
            // Set our state for the next step.
            Double nextState = new Double(next);
            gc.setNodeState(nextState);

            // Inform our neighbors for the next step.
            if (stepId != _steps-1) {
                if (nodeId != 0)
                    gc.sendMessage(nodeId-1, nextState);
                if (nodeId != gc.getNumNodes()-1)
                    gc.sendMessage(nodeId+1, nextState);
            }
            return new Result(nodeId, next);
        }
```

Example 12.7: Inform Neighboring Nodes, Set State (Continued)

```
    private static double toDouble(Object o) {
        if (o == null)
           return 0;
        else
            return ((Double) o).doubleValue();
    }

    private int _steps;
    private double _factor;
}
```

Result.java The Result class is a simple container for a node ID and a `double` value. Its `compareTo` method facilitates sorting by node ID.

Example 12.8: Result.java

```
package examples.simplepropagator;

class Result implements Comparable, java.io.Serializable {
    int nodeId;
    double value;

    Result(int n, double d) {
        nodeId = n;
        value = d;
    }

    public int compareTo(Object o) {
        return this.nodeId - ((Result) o).nodeId;
    }
}
```

Here is the output from the program:

Example 12.9: Program Output

```
0   10.00  0.00  0.00  0.00  0.00  8.00
1   5.00  5.00  0.00  0.00  4.00  4.00
2   5.00  2.50  2.50  2.00  2.00  4.00
3   3.75  3.75  2.25  2.25  3.00  3.00
4   3.75  3.00  3.00  2.62  2.62  3.00
5   3.38  3.38  2.81  2.81  2.81  2.81
6   3.38  3.09  3.09  2.81  2.81  2.81
7   3.23  3.23  2.95  2.95  2.81  2.81
8   3.23  3.09  3.09  2.88  2.88  2.81
9   3.16  3.16  2.99  2.99  2.85  2.85
10  3.16  3.08  3.08  2.92  2.92  2.85
11  3.12  3.12  3.00  3.00  2.88  2.88
```

Appendix A
Task Instrumentation

Introduction

This Appendix describes the instrumentation phases produced by enabling Task instrumentation. To enable Task instrumentation, see “Enabling Enhanced Task Instrumentation” on page 89 of the *GridServer Administration Guide*.

NOTE: Task instrumentation should be used for development purposes only, and not in production environments. It will slow down the Manager significantly, and also requires additional disk space, so it is important to disable it after you have completed using it.

All instrumentation phases have an absolute time marker, which is the time at the start of the action. Actions may also have a relative duration marker, if it is possible to measure the duration. The times are marked according to the client’s clock.

Instrumentation phases have the following syntax:

```
[Client] [Action] [Object]
```

Client

The Client of an instrumentation phase can be one of the following:

- Engine
- Driver
- Broker

Action

The Action of an instrumentation phase can be one of the following:

Action	Description
Send	A send of a message. The absolute time is the start time of the send, and there may or may not be a duration value.
Receive	A receive of a message. The absolute time is the end of the retrieval. There is no duration value.
Retrieve	A receive, with a measurement of the duration. The absolute time is the time at which the retrieval started.
Serialize	The conversion of an in-memory object to its serialized format, for transfer to another client.

Action	Description
Deserialize	The conversion of a serialized object to an in-memory object.
Write	The writing of data to a file, typically for DDT.
Download	The downloading of data from another client.
Call	A call to a user-implemented method.
Load	A native library load.

Object

The Object of an instrumentation phase can be one of the following

Object	Description
Jar	The JAR file, which is only used for dynamic class loading.
Instance	The instance object, which is either the tasklet or the initialization data.
Input	The input data or message.
Output	The output data or message
Update	The update data, message, or call
Checkpoint	Checkpoint data, if checkpointing is enabled.
Library	A native library
Initialize	The initialization call
Service	The service call
Completed	The callback on completion
Failed	The callback on failure
Serialize	The call to a user-implemented native serialize method
Deserialize	The call to a user-implemented native deserialize method

Phases

The following is the complete list of all phases.

Driver-side

Phase	Description
Driver Serialize Jar	The serialization of the JAR file when the JAR file is set.
Driver Serialize Instance	The serialization of the service instance object.

Phase	Description
Driver Serialize Input	The serialization of the service input.
Driver Send Input	The time of the Driver send of the input message to the Broker. Keep in mind that more than one input may be sent in one message.
Driver Call Completed	The callback of a successful task.
Driver Call Failed	The callback of a failed task.
Driver Download Output:	The download of output over DDT.
Driver Deserialize Output	The deserialization of output over DDT.
Driver Retrieve Output	The time at which the Driver receives the output message from the Broker. Keep in mind that more than one output may be retrieved in one message

Engine-side

Phase	Description
Engine Receive Input	The time at which the Engine receives the input message from the Broker.
Engine Download Instance	The download of the service instance object.
Engine Deserialize Instance	The deserialization of the service instance object.
Engine Call Initialize	The initialization call.
Engine Download Update	The download of update data.
Engine Deserialize Update	The deserialization of update data.
Engine Call Update	The update call.
Engine Download Input	The download of the input.
Engine Deserialize Input	The deserialization of the input.
Engine Download Checkpoint	The download of checkpoint data from another Engine.
Engine Call Service	The service call.
Engine Serialize Output	The serialization of the output.
Engine Send Output	The time at which the Engine sends the output message to the Broker.

Broker-side

Phase	Description
Broker Receive Input	The time at which the Broker received the input from the Driver.
Broker Send Input	The time at which the Broker sent the input to the Engine.
Broker Receive Output	The time at which the Broker received the output from the Engine.
Broker Send Output	The time at which the Broker sent the output to the Driver.
Broker Remove Output	The time at which the Broker removed the output due to the acknowledgement from the Driver.

DDT file writes

Phase	Description
[Client] Write Input:	The input file write.
[Client] Write Output:	The output file write.
[Client] Write Instance	The instance object write.
[Client] Write Jar	The JAR file write.
[Client] Write Update	The update data write.

Native

Phase	Description
Engine Load Library	The load of a native dynamic library.
Driver Call Serialize	The native object serialize call.
Driver Call Deserialize	The native object deserialize call.

Appendix B
SOAPActions

This section is provided as a reference for `SOAPActions` defined for Services exposed as Web Services. Since the actions are automatically attached to the associated operations, it is not necessary for a user of the Service to know the meaning of each action.

Action	Description
`init`	Indicates that this operation will create a new session. The return value of the operation is the URL if the new instance. The name of the operation has no meaning.
`destroy`	Indicates that this operation will destroy the session. If this session is the default session, a `SOAPFault` is generated. The name of the operation has no meaning.
`appendState`	Indicates that this operation is an append state operation. The operation name must match the method name.
`setState`	Indicates that this operation is a set state operation. The operation name must match the method name.
`submit`	Indicates that this operation is an asynchronous submission. The return value is the ID of the session. The name of the operation must be `[method name]_Async`.
`collect`	Indicates that this operation should collect the result of the given ID, or the next available if the given ID is null. The name of the operation has no meaning.
[no action]	Indicates that the operation is a synchronous operation. The operation name must match the method name.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

V

W

X