SIMPROCESS®
and
Dispatcher

SIMPROCESS as a Service

SIMPROCESS and Dispatcher

Table of Contents

SIMPROCESS and Dispatcher

1 Introduction

SIMPROCESS® has a proven track record in its role as a modeling and simulation tool. More and more often, however, it's useful to be able to call on simulation capabilities and use the results to make automated business decisions. SIMPROCESS can now provide that capability through the DispatcherService, a Web service that can be deployed in a Web Container (an application supporting the Java Servlet Specification) or an Application Server, and the SIMPROCESS Dispatcher. This document provides details on the components required to use this capability.

2 SIMPROCESS and Dispatcher Installation

In order to use the Dispatcher and related features, an alternate install set is offered by the installer, called "SIMPROCESS and Dispatcher." This is not the default set and must be specifically selected. It provides all the components needed to use SIMPROCESS as a regular desktop application, but it also includes new components which allow use of the Dispatcher to invoke SIMPROCESS as a service. The key additions are described here.

- **Dispatcher** is the new program (it will have the.*exe* extension on Windows systems) that controls the operation of one or more instances of SIMPROCESS when requests are received via the DispatcherService.
- **dispatcher** is the directory containing all of the other components needed to run SIMPROCESS as a service (including the DispatcherService deployment files). Its precise contents will vary according to the platform on which installation occurs. More detailed information on some of these components will appear later to explain their configuration and operation.

Installation is but one of the steps that enable SIMPROCESS to operate as a service. The DispatcherService, included among the components in the new *dispatcher* directory, will be described in detail later. It is the gateway through which a client application obtains SIMPROCESS services. A Web Container or Application Server will be required in order to deploy and use the DispatcherService. A client application must be available that knows how to communicate with the DispatcherService. A Dispatcher must be able to advertise that its services are available to a DispatcherService. The DispatcherService must know where to look for the list of service providers. And finally, a Dispatcher must be able to obtain licenses when services are requested.

The next section provides information on licensing for the Dispatcher. The remaining sections provide information on the configuration and operation of all other components.

3 Licensing

The licensing of SIMPROCESS as a desktop application is described in Chapter 2 of the *SIMPROCESS Getting Started* manual. SIMPROCESS can still be used that way with the additional Dispatcher-related components installed (although running it at the same time that the Dispatcher is responding to requests is not recommended). The Dispatcher, however, is designed to operate as a server application which can start as many instances

of SIMPROCESS as it is licensed to support. For that reason, its licenses must be served by a FLEXlm license server.

3.1 The License Server

The components required to set up and operate a FLEXlm license server are not included with the installer. Instead, they will be provided separately by your SIMPROCESS technical support team. This gives you the flexibility to decide where you want to place your license server.

Although FLEXlm license servers are available for a variety of platforms (combinations of operating systems and processors), it is only possible to serve SIMPROCESS licenses on a platform for which SIMPROCESS itself is available. All binary files required for a license server can be downloaded from Macrovision except for our "vendor daemon" program, *dcacisim*, and we can only compile it for those platforms on which we currently offer SIMPROCESS. However, it is possible to mix and match among these platforms to suit your needs. That is, you might wish to run the Dispatcher on a Linux system (where performance will often be significantly better than on Windows) and place your license server on a Windows platform. Or you may wish to run multiple Dispatcher installations, perhaps even on mixed platforms, sharing a pool of licenses from a single license server.

The available options provide you with greater flexibility and control over distribution of the workload. Your SIMPROCESS technical support team will help you choose a license server strategy that best suits your operational needs.

3.2 The license.dat File

The Dispatcher application must obtain a license before it can launch an instance of SIMPROCESS when services are requested. The information on how to contact the license server is contained in a license file that must reside in the *dispatcher* directory. It should be a copy of the *license.dat* file used by the license server, and it must also have that name.

To obtain a license file, you must provide the SIMPROCESS Help Desk with the *hostid* value of the system on which your FLEXlm license server will run. If the server will be on the same operating system as your Dispatcher, you can use the *hostid.bat* or *hostid.sh* included in the root of the SIMPROCESS installation directory to put this value into the *hostid.txt* file. If not, the Help Desk can provide you with a copy of the *lmutil* program for the license server's particular platform and instructions on how to have it put your *hostid* value in a file that you can send to them to obtain a license file.

Once you've received the *license.dat* file, you'll need to know that only one portion of its contents can be edited without invalidating the contents. The first line of the file will look something like the following sample:

```
SERVER 192.168.30.30 0060970572a5
```

First is the keyword SERVER, which must never be changed. The last part will be the *hostid* value that was provided to the Help Desk to obtain the license file; it must also never be changed. The middle part, however, is the Internet Protocol (IP) address of the system on which the license server is to run. Because network configurations sometimes change, this value can be changed if required. If possible, however, it's usually a best to use a name in this position (e.g., myhost.mycompany.com) rather than an actual numerical IP address, and ensure that all installed Dispatcher applications communicating with this license server can resolve the name to the correct machine. On the server itself, it's allowable to replace the IP address with the keyword *this_host*, which eliminates the need to change the file should the system's IP address change. If the license server will reside on the same system as a Dispatcher, then that Dispatcher's *license.dat* file may also use the *this_host* keyword. Any other changes in the license file will render it invalid and the Dispatcher will be unable to obtain licenses to run SIMPROCESS instances.

4 The RegistryServer Application

The RegistryServer is the smallest piece of the complete puzzle, but without it nothing else will work. The RegistryServer is the means by which a Dispatcher advertises that its services are available. It's the means by which a DispatcherService locates available Dispatchers to obtain SIMPROCESS services. It provides a registry – a place where other things can deposit and look up information. Therefore, the RegistryServer must be running in order for any Dispatcher to register itself as an available service provider. In addition, the DispatcherService must know where it can be found so that it can obtain a list of available service providers. Later sections contain further configuration details for those components.

4.1.1 Configuring RegistryServer

The *registry.properties* file must be in the same directory with *dispatcher.jar* for proper operation. In most situations, the original contents of this file will work properly on any system without changes, but it may be necessary to modify specific entries.

- The line "java.naming.provider.url=rmi://localhost:5700" should only be changed if port 5700 is in conflict with another application on the network. This port was selected because it is not reserved for use by any well known network service. If it is necessary to change this port, it's important that the same value be used in the properties files for both the Dispatcher and the DispatcherService so that both will know how to find the RegistryServer application. *Do not change any other portion of this line.*
- The line "#registry.logging=true" should not normally be enabled. However, if there is a significant problem with network communications, enabling this line by removing the "#" will cause RegistryServer to log all communications. This will produce a large amount of output, so it should only be done if absolutely essential. Enabling this line while RegistryServer is running will **not** enable logging.
- The line "#registry.debugging=true" can be enabled by removing the "#" at the beginning if debugging output is desired. This is normally not required except perhaps during initial deployment testing. In normal usage (i.e., with this line disabled), there will be very little output placed into the log and error files. When

enabled, some of the internal logic will display information into the log file about calls received and information provided in response. Enabling this line while RegistryServer is running will **not** enable debugging output.

4.1.2 Running RegistryServer

The RegistryServer application is specifically designed not to depend on any particular hardware or operating system. It only requires that the system on which it runs has a Java Runtime Environment (JRE) that is fully compatible with Sun's Java 1.4.1 or newer. Therefore, it does not have to be run within the confines of the SIMPROCESS directory structure, and it may be hosted on any available system with a compatible JRE. The installer provides the files *RegistryServer.sh* and *RegistryServer.bat* in the *dispatcher* directory, which are preconfigured examples of how to run it using the JRE installed with SIMPROCESS. These can be used as examples for running RegistryServer on any system with a compatible JRE. To run RegistryServer, its two required files must be in the same directory:

- dispatcher.jar
- registry.properties

RegistryServer has been tested on Windows 2000, Windows XP Professional, Windows 2003 Server, Mac OS X 10.3 and Linux Mandrake 9.1. On Mac OS X the test used Apple's Java 1.4.2 JRE, while the others were tested with a JRE obtained from Sun Microsystems at http://java.sun.com.

4.1.3 Running RegistryServer as a Windows Service

It will usually be preferable to run RegistryServer so that it requires no user intervention. That is, it should be able to start up and run automatically without a user session being kept active to support it. On Windows systems, this can be accomplished by installing it as a Windows Service.

The "SIMPROCESS and Dispatcher" install set will include all the files needed to do this (even if installing on a non-Windows platform). If you wish to install RegistryServer as a Service on a Windows system other than the one on which the Dispatcher will operate, the files described in the following paragraphs will need to be copied to the same directory as the *dispatcher.jar* and *registry.properties* files described above, though one of them will require additional changes to be made. Here are the files provided, along with instructions on their use.

4.1.3.1 RegistryService.exe

This executable file will install the Windows Service using parameters passed to it by the RegistryServiceInstall.bat file.

4.1.3.2 RegistryServiceInstall.bat

This batch file executes the RegistryService.exe program and passes parameters that will cause it to install a Service named "SIMPROCESS RegistryService" that will start automatically each time your system starts up. As initially provided by the installer, it will only work correctly when run from the dispatcher directory of SIMPROCESS. To

install RegistryServer as a Service on another Windows system, this file must be edited. The seven steps defined in comments in the original will serve as the basis for explaining what changes must be made there.

- Steps 1 and 2 are used in the installed SIMPROCESS setting to obtain the complete directory name where SIMPROCESS was installed. These can be removed in a copy to be used on another system. Alternatively, you can change either or both of them to provide information required in later steps.
- Step 3 sets the variable ***JVMLoc*** to the complete path of a file named *jvm.dll* included with the JRE for running RegistryServer. For the modified copy of this file, change this line to provide the full path to that file in the JRE to be used. For example, if the system contains Sun's Java Plug-in for use with applets in web browsers, there will be a JRE included in it. The complete path on a typical system might be *C:\Program Files\Java\j2re1.4.2_05\bin\client\jvm.dll*. It's also important to note that a path containing spaces (like this one does) must be enclosed in quotes. So the resulting line might look like this:

  ```
  set JVMLoc="C:\Program Files\Java\j2re1.4.2_05\bin\client\jvm.dll"
  ```

- Step 4 sets the variable ***WorkDir*** to the path of the "working directory" where RegistryServer will run. This will allow it to find its *registry.properties* file, and is where its log and error files will be created (named *DispatcherRegistry.log* and *DispatcherRegistry.err*). Just as with ***JVMLoc***, this requires quotes if any spaces appear in the name (they are allowed even if there are no spaces). The directory can be any place you choose, so the command might look like this:

  ```
  set WorkDir="D:\work\registryserver"
  ```

- The remaining steps *must not* be changed. Note Step 6, however, which actually installs the new Service using the name "SIMPROCESS RegistryServer." If no problem occurs, the system will display a message saying that the installation of the new Service was successful. Step 7 then reminds you that the newly installed Service must be started, or that it will start automatically upon restarting your system.
- To start the new Service once installation is accomplished, you can go to the Windows Control Panel and select Administrative Tools, then Services. Select the Service to start or stop it as needed.

4.1.3.3 RegistryServiceUninstall.bat

This batch file uninstalls the Windows Service named "SIMPROCESS RegistryService." It will fail if the Service is currently running. No changes to this file will be required. It must reside in the same directory with *RegistryService.exe*.

4.1.4 Running RegistryServer as a Server on Other Systems

Java programs have a tendency to shut themselves down when any kind of user logoff action occurs. As a result, RegistryServer couldn't simply be run as a Scheduled Task in Windows since it would terminate if any user logged off the system at the completion of

some task. A similar problem can arise on Unix-based systems (which includes Linux, Mac OS X, and many others), so that simply running the program in the background by appending an ampersand ("&") to the command will not necessarily work. Here are two alternatives which might be useful to deal with this issue when running RegistryServer on these systems.

- Use the *nohup* command. This is usually done by preceding the command with the word "nohup" and following it with an ampersand to put it in the background (for example, *nohup ./RegistryServer.sh &*). When using *nohup*, a program will not stop when the user logs off the system ("hangs up"). Instead, it continues to run in the background. This may not be an ideal solution, however, depending on the method used to connect to the system, as it sometimes prevents proper termination of X-server sessions or has other undesirable side effects. It will also require that you repeat the process after a system restart.
- Add the appropriate command to a system startup file. (Precise details will vary with the operating system.) Some Linux systems, for instance, look for a file named */etc/rc.d/rc.local* to be executed during the system startup process. Putting an entry here to start the RegistryServer will ensure that it's run every time your system starts, which guarantees that it will be available. So you might make an entry that will run a shell script or Java command line as a non-privileged user (i.e., a user other than root – you should never use root for such purposes) something like this:

```
/bin/su username -c "cd dirname;./scriptname.sh"
```

4.1.5 Shutting Down the RegistryServer Server

Sometimes you'll want to shut down the RegistryServer without shutting down the entire system. If you installed your RegistryServer as a Windows service, instructions are provided above for shutting it down. If you've set it up to run as a server on a non-Windows system, read on. In either case, it's wise to ensure that any Dispatchers which refer to it are shut down first. Shutting them down lets them remove their entries from the RegistryServer and clean up, rather than being orphaned and shutting down due to a communication failure.

When shutting down RegistryServer on Unix-based platforms, you'll want to understand the proper (and safe) method of shutting it down. The key is to find the first "java" process in the set of processes that were created by starting it. Look at a detailed process list using a form of *ps* appropriate to the operating system and trace the Process ID (PID) values to find the first one in the chain. If you've used a shell script file, you'll want to use the first "java" process owned by it and not the PID of the shell command itself. Now you need to stop that process with the *kill* command and an appropriate signal. Many signals will either have no effect, or may result in an improper (and thus unsafe) shutdown. To ensure that RegistryServer is able to shut down cleanly, you should always use SIGINT, which is an interrupt signal. If RegistryServer.sh were run from a command line, using the Ctrl+C key combination delivers a SIGINT to the program, which lets it shut down cleanly. Sending SIGINT with the *kill* command does the same when it's in

the background. You may need to use *man kill* to learn the appropriate form of the *kill* command to use on your particular system.

5 Dispatcher

The Dispatcher has previously been addressed in terms of licensing, but that omitted the details of its configuration and operation. Now it's time to fill in those details.

5.1.1 Locating a RegistryServer

As was noted earlier, the RegistryServer is a critical part of providing SIMPROCESS services. A Dispatcher must be able to communicate with a RegistryServer on startup to list itself as a provider of SIMPROCESS services. Its configuration must tell it where to find a RegistryServer, and it must be able to successfully communicate with it in order to function. If it cannot do so, it will shut itself down. The information required to locate a RegistryServer is found in the file named *dispatcher.properties*, which the installer places in the *dispatcher* directory. This file must always be in this directory. Here is the portion of the installed file that tells a Dispatcher how to find its RegistryServer:

```
#
# Naming provider URL
#   If RegistryServer is not running at port 5700, change the port number
#   in the line below to match.
#
#   Determine the hostname or IP where RegistryServer is running if not
#   on the same system with the Dispatcher and replace "localhost" with
#   that value.
#
java.naming.provider.url=rmi://localhost:5700
```

Much of this is comment text explaining what the property must contain. There are only two specific parts that can be changed without rendering the file useless: (1) *localhost* and (2) *5700*. The first item, *localhost*, is a universal name which refers to the same computer on which the Dispatcher application is running. If RegistryServer is running on the same system as the Dispatcher, this value can remain unchanged. If not, it should be changed to an IP address or to a name that the system can resolve to the IP address of the system where RegistryServer is running (e.g., myhost.mycompany.com). The second item is the network port over which communication will occur between the Dispatcher and RegistryServer applications. This should only be changed if the port for RegistryServer was changed in its *registry.properties* file.

Important: The Dispatcher *requires* a RegistryServer to be reachable upon startup. Therefore, if a RegistryServer cannot be reached using the values in this properties file when starting up, it will shut down immediately and record the trouble in its output file(s). Always make sure the RegistryServer is available before starting the Dispatcher.

5.1.2 Dispatcher Debugging

Only one other property in the *dispatcher.properties* file should ever be changed. The last line in the installed file reads "#dispatcher.debugging=true" and should ordinarily be left alone. However, if additional information is desired in the Dispatcher's log and error files (or its window when running the application directly, as described later), the "#" can

be removed at the beginning to enable debugging output. This puts a significant amount of output into the log and error files (*Dispatcher.log* and *Dispatcher.err*) that is primarily only useful to developers. Enabling it is not generally recommended.

5.1.3 Locating a License Server

A Dispatcher does not depend upon being able to communicate with a FLEXlm license server at initial startup. However, it must be able to check out a license when asked to provide SIMPROCESS services. If it cannot, it is unable to render the requested services and responds accordingly. For a Dispatcher to provide services, it must (1) be able to communicate with the FLEXlm license server identified in its *license.dat* file, and (2) the license server must have an available license when requested. If it cannot contact the license server, or if the server is unable to grant a license request for any reason, then the Dispatcher will reply that it is unable to provide the requested services.

5.1.4 How Many Dispatchers?

It is possible to run more than one Dispatcher from a single SIMPROCESS installation. However, because of the way that each one writes to its log and error files, this would result in sharing of files and is generally not recommended. On a limited basis, it may be useful for initial testing, but should otherwise be avoided.

To run more than one Dispatcher on a single computer system, install multiple copies of SIMPROCESS. Each Dispatcher will have its own properties file, and each will write to separate log and error files, eliminating any potential conflicts. Each must also have its own *license.dat* file to enable it to communicate with a license server.

Windows systems typically don't perform sufficiently well to make multiple installations practical, but Linux systems are ideal for such purposes. They tend to do much better at memory management and are better suited to running large numbers of applications simultaneously. If a Windows system is to host multiple Dispatchers, it should have the fastest possible processor and the maximum possible amount of RAM per SIMPROCESS license. In addition, if run as a Service (see below), there may be special considerations needed in setting up each one.

5.1.5 Application or Server?

There are two distinct ways to run a Dispatcher. One is as an application, where a user starts the program and it shows a window. The other is as a server, where the program runs without user interaction and will typically be able to run with no user logged on.

5.1.5.1 Application

To run the Dispatcher as an application, first make sure that all previously mentioned criteria have been met. That is, make sure that the required information is provided to communicate with a FLEXlm license server and RegistryServer as needed. Then run the Dispatcher program found in the root of the SIMPROCESS installation directory. When run in this way, the Dispatcher will display a window. This window will display a message at startup indicating the internal name by which it will be known (this name is only meaningful to the RegistryServer). During Dispatcher operations, it will display

occasional messages indicating that it has received and responded to requests for services. (There will be much more information in this window and the log and error files if debugging is enabled.) Running a Dispatcher as an application can be useful for becoming familiar with how it works, for testing situations where multiple Dispatchers may be helpful, or a variety of other purposes. A Dispatcher run in this manner can be shut down by using its File menu, or simply by closing its window. When this occurs, it will prompt for confirmation if any SIMPROCESS instances are still running. Instances must periodically confirm that they can still communicate with their owning Dispatcher and will shut down when that fails. A Dispatcher controls the licenses it checks out, so its termination frees all licenses it holds and leaves instances without valid authorizations.

5.1.5.2 Server

The most common use of a Dispatcher will be in a setting where it can run with no user interaction (i.e., no user session is needed) and it will not be hampered by users logging on to perform some task and then logging off again. Just how this is accomplished will depend on the platform where it will be running.

5.1.5.2.1 Running Dispatcher as a Windows Service

When installed on a Windows system, the *dispatcher* directory contains files that allow the installation of a Windows Service that will automatically run each time the system is booted. These files are described below.

- *DispatcherServer.exe* is the executable program which installs a Windows Service named "SIMPROCESS Dispatcher" using the additional parameters provided by a batch file.
- *DispatcherServerInstall.bat* is a batch file that executes *DispatcherServer.exe* and provides it the necessary parameters.
- *DispatcherServerUninstall.bat* is a batch file that executes *DispatcherServer.exe* to uninstall the previously installed Windows Service named "SIMPROCESS Dispatcher." Note that the service must be stopped before it can be successfully uninstalled.

These files must remain in the directory where they are installed. Only the file named *DispatcherServerInstall.bat* should ever be modified in any way, and then only under certain conditions. Specifically, if the Dispatcher is to run as a Service on the same system where a "SIMPROCESS RegistryServer" will also be installed, it's advantageous to change this file so that a dependency is established between the two. To do that, find the following lines in the *DispatcherServerInstall.bat* file:

```
REM If dependency on the SIMPROCESS RegistryServer service
REM is desired, uncomment the following line.
REM set DEPENDS=-depends "SIMPROCESS RegistryServer"
```

On the last of those lines, remove the "REM" and the space which follows it, then save the modified file. This will cause the new Service named "SIMPROCESS Dispatcher" to be dependent on the "SIMPROCESS RegistryServer" so that the latter will automatically start if it's not running.

Now simply double-click the batch file to install the Service. A reminder will appear stating that you'll need to start the Service manually unless you wish to restart your system. To start manually, use the Windows Control Panel and choose Administrative Tools, then Services. In the list of Services, select the "SIMPROCESS Dispatcher" and start it. If the dependency is set, this will also start "SIMPROCESS RegistryServer" if necessary.

5.1.5.2.2 As a Server on Other Systems

As stated earlier, Java programs have a tendency to shut themselves down when any kind of user logoff action occurs. So on Unix-based platforms, simply running the Dispatcher in the background by appending an ampersand ("&") to the command will not work. Here are two alternatives which might be useful for running Dispatcher as a server on these systems.

- Run using the *nohup* command. This is usually done by preceding the command with the word "nohup" and appending an ampersand to put it in the background (e.g., *nohup ./Dispatcher svc &*). When using *nohup*, a program will not stop when the user logs off the system ("hangs up"). Instead, it continues to run in the background. This may not be an ideal solution, depending on the method used to connect to the system, as it sometimes prevents proper termination of X-server sessions or has other undesirable side effects. It will also require manually starting the Dispatcher via the *nohup* command any time the system restarts.
- Add the appropriate command to a system startup file. (The precise details will differ according to the actual operating system.) Some Linux systems, for instance, look for a file named /etc/rc.d/rc.local that will be executed during the system startup sequence. So you might make an entry that will run a shell script as a non-privileged user (i.e., a user other than root – you should never use root for such purposes) like this:

```
/bin/su username -c "cd installdir;sh Dispatcher svc"
```

Remember that the earlier discussion of running a Dispatcher as an application said that it displays a window for user interaction. Clearly, that is not a good thing when running as a server. For that reason, the Dispatcher is designed to alter its behavior when receiving a command line parameter. In the two examples shown above, a parameter value of *svc* was included on the command line. This signals to the Dispatcher that it is being run as a server and suppresses the creation of its window. Failure to include this parameter will not produce the desired results.

Important Note: When placing entries in system startup files for both a RegistryServer and one or more Dispatchers on one system, be sure to start the RegistryServer first so that each Dispatcher will be able to communicate with the RegistryServer.

There may be times when it's necessary to shut down such a Dispatcher without shutting down the entire system. There are two methods that may be used.

- Shut down the RegistryServer on which the Dispatcher depends. Each Dispatcher periodically checks to determine if its RegistryServer is available. If it is not able

to communicate with its RegistryServer, it will shut down. This may not be a good choice if more than one Dispatcher uses the same RegistryServer. Since the RegistryServer may not reside on the same system (perhaps not even the same operating system) where a Dispatcher runs, refer to the appropriate instructions for how to shut it down.

- The preferred way to shut down a Dispatcher running as a server is to identify its Process ID (PID) number and use the *kill* command. Only send the signal called SIGINT, since any other will either be ignored or may not allow the Dispatcher an opportunity to properly clean up (by removing its entry from the RegistryServer, for instance). Finding the right PID can be a little tricky, and the precise form of the *ps* command needed can vary among operating systems. When started with a command like the example shown above, look for processes whose command lines include the complete path to the installed JRE's "java" command (e.g., */installdir/jre/bin/java*) followed by "com.zerog.lax.LAX". Select the first PID in the sequence and send it a SIGINT to interrupt it. You may want to use *man kill* to determine the correct form of the command for your system. The Dispatcher will then shut down just as if it had been running as an application with a window and the File menu was used to stop the program.

6 DispatcherService

The DispatcherService is the gateway to SIMPROCESS services for client applications. Naturally, it will need to know about a RegistryServer to get a list of service providers. And it will communicate with a Dispatcher registered there to obtain the services. But all that is the magic behind the curtain. The client, generally speaking, will only know about the DispatcherService. It will never interact directly with the other players. This section tells you how to put the DispatcherService to work for a client.

6.1 Installed DispatcherService Components

In the *dispatcher* directory is a subdirectory named *DispatcherService*. It contains the components you must deploy to provide SIMPROCESS services to client applications. The files installed are listed below.

- **Dispatcher.wsdl** is the Web Services Description Language (WSDL) file which describes the operations available from the deployed DispatcherService Web service. It's not necessary to have this file to deploy the service, since you'll be able to obtain one from your Web Container or Application Server, but for the technically curious who want a head start, it can help in planning for the development of a client application.
- **Dispatcher-jaxrpc.war** is a Web Archive (WAR) file containing the complete, deployable DispatcherService. As its name implies, it was developed in Java using Sun's Java APIs for XML RPC (JAX-RPC). This isn't really critical to know for a client application developer, but technical people may recognize some of the WSDL file's contents more readily once they're aware of this fact.
- **dispatcherservice.properties** is a properties file similar to those used by both the Dispatcher and RegistryServer programs. It must be deployed along with the WAR file so that the DispatcherService can find the location of a RegistryServer

to obtain a list of service providers, along with some other configuration options described in the next paragraph.

6.2 Configuration Options

The *dispatcherservice.properties* file contains five items which can be set to affect its operation.

6.2.1 Locating the RegistryServer

Just as with a Dispatcher, the DispatcherService must be able to find a RegistryServer that has a list of service providers (Dispatchers). The properties file contains an entry for this purpose, which initially appears as follows:

```
#
# Naming provider URL
#    Determine the hostname or IP where RegistryServer is
#       running and use that instead of localhost.
#    If this property in the RegistryServer's properties
#       file is not using port 5700, change the port value
#       below to match its value.
#
java.naming.provider.url=rmi://localhost:5700
```

Only if the RegistryServer has used a port number other than 5700 in its own properties file should that portion of this property be changed. The hostname *localhost* should remain only if the RegistryServer is on the same system as the Web Container or Application Server where you'll deploy the DispatcherService. Otherwise, you should change it to an IP address or hostname that will locate that system on the network.

6.2.2 Other Property Settings

The *dispatcherservice.properties* file contains a comment block indicating that all items below it are configurable by the individual responsible for configuring and deploying the DispatcherService. With the exception of the single item above, nothing else above this comment block should be changed in any way. Here are the remaining configurable properties and their uses.

```
#
# Default timeout of instance in minutes.  Must be positive.
#
instance.timeout.default=2
```

The default instance timeout is the default amount of time that can elapse without commands being sent to a SIMPROCESS instance before it will be considered idle. When an instance is started by a client application sending an appropriate message to the DispatcherService, it will use the timeout value provided if allowed. If the value provided is non-positive or greater than the maximum (see below), this default value will be used instead. Each time the DispatcherService's timer fires, it will identify any instance deemed idle and ask its Dispatcher to shut it down, thereby freeing a license. If the instance is running a simulation or actively carrying out some other long-running

request, it will reject the request to shut down. It's best to use a relatively small default timeout value to prevent the possibility of an idle instance needlessly tying up a license.

```
#
# Maximum timeout of instance in minutes.  Must be positive.
#
instance.timeout.max=60
```

The maximum instance timeout is the largest timeout value that can be used when starting an instance or by sending a message to set an instance's timeout value. If any message includes a timeout value larger than this, that value will be discarded and this one used instead. This is useful to prevent an idle instance from unnecessarily tying up a license.

```
#
# Timer interval for checking timeouts (in minutes).
# Must be positive.
#
timer.interval=2
```

This property determines how frequently the DispatcherService's internal timer will fire. Each time it fires, its list of instances will be checked to determine which ones have not had any messages delivered for a sufficient period of time to exceed their timeout limits. When one is found, its Dispatcher will be asked to shut it down. If it's actively engaged in receiving another command or simulating, it will be disregarded until the next timer firing occurs. If it responds that an error has occurred, its Dispatcher will be asked to forcefully shut it down. Setting too small a value for this property can result in too many requests for instance shutdown, thereby risking the integrity of communications with any given Dispatcher. Using too large a value can result in extended idle times that needlessly tie up licenses.

```
#
# Debugging (should normally be disabled)
#
#service.debugging=true
```

The service.debugging property is initially disabled. To enable it, remove the "#" from the beginning of the line. When enabled, the DispatcherService will write output into the standard output and standard error files of the Web Container or Application Server where it operates. The names and locations of those files depend on the product being used. When debugging has been enabled, the service will display an indication when it's loaded for the first time as it constructs its timer, including displaying its timer interval. It will also display messages at the start of its timer logic and then again at the end, along with possible messages in between if it finds timed out instances and sends messages to have them stopped.

6.2.3 Changing Properties after Starting DispatcherService

The DispatcherService reads its properties once when initially loaded by the Web Container or Application Server where it's deployed. If it becomes necessary to change

any of the properties in this file after loading, the service usually must be stopped and restarted in order to have it read the newly modified file. The specific mechanism for starting, stopping or restarting a Web service depends on the particular Web Container or Application Server used, as do the details of where the *dispatcherservice.properties* file should be placed.

6.3 Deploying the DispatcherService

The Dispatcher-jaxrpc.war file is self-contained (except for the properties file described above) and ready to deploy into a suitable Web Container or Application Server. The selected Web Container or Application Server must include support for the following technologies:

- Java Servlet 2.3 specification
- Java API for XML-based RPC (JAX-RPC)

It must also include support for the Java API for XML Processing (JAXP), version 1.2 or higher. JAXP 1.2 is included in Sun's Java 2 Standard Edition (J2SE) version 1.4. JAXP and JAX-RPC support are both included with the Java 2 Enterprise Edition (J2EE) 1.4. A Web Container or Application Server supporting the J2EE 1.4 specification will meet these requirements.

Development and testing of DispatcherService was done using a modified version of the Apache Jakarta Tomcat servlet container supporting the required technologies. For its first real world deployment, the customer chose to use Tomcat 5. So the ultimate choice of Web Container or Application Server may depend very much on the specific needs of the client application(s) which will interact with it. Thus the candidates listed here may not include one that is suited to a particular environment or purpose, nor should the list be considered exhaustive. Also, none of these products should be considered to be endorsed by CACI or in any way recommended. This list is provided as a convenience only and contains information which may assist in the selection of a product.

PRODUCT	TYPE	LICENSE	ADDITIONAL COMMENTS
Tomcat	Servlet Container	Apache 2.0	Used in the official Reference Implementation for the Java Servlet and JavaServer Pages specifications; can be integrated with a web server if required
Caucho Resin	Application Server	GPL	Can be integrated with a web server if required; commercial version available
Sun Java System Application Server	Application Server	N/A	Purchased support available
JBoss Application Server	Application Server	LGPL	J2EE 1.4 certified; bundles Tomcat to provide Servlet and JSP support

There are many others available, including commercial products. For additional information and possible links to other resources which can help in choosing a suitable product, visit TheServerSide Application Server Matrix. This site has a longer list of products along with links to reviews, vendor web sites, and more.

Once a product is installed and configured, you will need to take product-specific actions to deploy the *Dispatcher-jaxrpc.war* and *dispatcherservice.properties* files. Tomcat includes a "Manager Application" that can be used to stop, start, deploy or undeploy Web services and applications, including uploading WAR files for deployment. The Tomcat

documentation also designates the location of "resources" needed by a service or application which aren't already built into it. The *dispatcherservice.properties* file is such a resource and must be placed in the *<Tomcat-installDir/>*shared/classes directory in order to be found by the DispatcherService when it initially loads. Similar information will be documented for the selected Web Container or Application Server.

6.4 DispatcherService Operations

Precise details on the use of the DispatcherService to obtain SIMPROCESS services will depend on numerous factors, including the Web Container or Application Server chosen for its deployment and the nature of the client application. Client applications might take the form of JavaServer Pages (JSPs). They could be Web services themselves. Or, they might be standalone programs written in any of a variety of programming languages. That's a key reason that the WSDL file, describing the DispatcherService in a way that is meaningful to application developers and their tools, is included among the installed files.

Equally important is an understanding of how the available operations might be applied to get the benefits of using SIMPROCESS simulation services on demand. So before listing the available operations, let's first look at some scenarios outlining how they might be used to satisfy the needs of a client application.

6.4.1 DispatcherService Usage Scenarios

A list of available operations will be more meaningful if they can be understood as a set of actions needed to satisfy a business requirement. The scenarios that follow describe some situations in which simulation services are important to a business process and show how SIMPROCESS can be used on demand to can satisfy that need. Each will generically present a situation, followed by a series of steps which can be taken to obtain SIMPROCESS services via DispatcherService. Each step will provide the name *in italics* of one or more operations which will be described in greater detail in the next section listing available operations. These scenarios leave out the handling of error conditions, which would be essential activities in a client application.

6.4.1.1 Scenario 1

A client application which monitors ongoing activities in a business process needs to respond to certain infrequent events by simulating a model using SIMPROCESS. The operations called on by the client application might be:

- Request the start of a SIMPROCESS instance, optionally providing a positive timeout (*startInstance*)
- Ask the instance to open a model (*openModel*)
- Ask the instance to set some parameters in the model (*setModelParameters*)
- Ask the instance to begin simulating the model (*startSimulation*)
- Check periodically until the simulation is complete (*isSimulationComplete* or *getSimulationStatus*)
- Optionally close the model (*closeModel*)
- Stop the instance (which will close any open models), or allow the timeout to expire if it is reasonably short, to free the license (*stopInstance*)

6.4.1.2 Scenario 2

At specific timed intervals, an application needs to simulate a model and provide results to users. The operations called on by the client application might be:

- Request the start of a SIMPROCESS instance (*startInstance*)
- Set the timeout on the instance to the maximum allowable value, so that the license is not released to other client applications (*getMaxTimeout*, *setTimeout*)
- Ask the instance to open a model (*openModel*)
- Ask the instance to set some parameters in the model (*setModelParameters*)
- Ask the instance to begin simulating the model (*startSimulation*)
- Check periodically until the simulation is complete (*isSimulationComplete* or *getSimulationStatus*)
- Repeatedly set parameters, simulate the model, and check status as needed based on some business rules in the client application (*setModelParameters*, *startSimulation*, *isSimulationComplete* or *getSimulationStatus*)
- At application shutdown, stop the instance (which will close any open models) and free the license (*stopSimulation*)

6.4.1.3 Scenario 3

An interactive, Web-based client application needs to pass some of the user-entered data to a model to use as parameters and simulate. Once simulation completes, it needs to report to users the results of the simulation when available, after which they may elect to modify their entries and repeat the simulation process. Once the user has entered all client-edited information, the operations called on by the client application might be:

- Request the start of a SIMPROCESS instance (*startInstance*)
- Ask the instance to open a model (*openModel*)
- Ask the instance to set some parameters in the model, using data entered by the user (*setModelParameters*)
- Ask the instance to begin simulating the model (*startSimulation*)
- Check periodically until the simulation is complete (*isSimulationComplete* or *getSimulationStatus*)
- Stop the instance (which will close any open models) and free the license (*stopSimulation*)

Clearly, many scenarios will take very similar steps to satisfy their requirements. The powerful features of SIMPROCESS allow tremendous flexibility in reading and writing databases or taking other actions which facilitate information exchange between the model being simulated and the client. In the next section, you'll find a list of all the operations offered by DispatcherService to implement these actions, as well as proper error handling, so that a client can take advantage of the full power of SIMPROCESS simulation services at need.

6.4.2 Available DispatcherService Operations

Because the DispatcherService was developed in Java using the JAX-RPC and JAXP technologies, the simplest way to describe the available operations is as Java methods. The paragraphs which follow will show each available operation as a Java method, along

with a brief description of its purpose and resulting error settings for the service and any affected SIMPROCESS instance. Those needing to know greater levels of technical detail (concerning SOAP message formats, encoding style, operation style, etc.) should refer to the WSDL file.

6.4.2.1 Say Hello

Method	public String sayHello()
Description	Ask DispatcherService to say hello
Error settings after call	Service: NOERR (List of errors is in Appendix A) Instance: N/A
Special Conditions	None
Usage Comments	Mainly useful as a test of successful communication, returns the literal string "Hello from the DispatcherService"

6.4.2.2 Check for Error

Method	public boolean error()
Description	Determine if DispatcherService error occurred
Error settings after call	Service: Not changed Instance: N/A
Special Conditions	None
Usage Comments	Call after most operations to determine if a non-zero error (any error other than NOERR) exists in DispatcherService's error manager.
Alternate Operation	**public String errorStr()** invokes this method and returns the string containing "true" or "false".

6.4.2.3 Last Error Number

Method	public int lastError()
Description	Get DispatcherService's last error number
Error settings after call	Service: Not changed Instance: N/A
Special Conditions	None
Usage Comments	Call after most operations to obtain the current error number value from DispatcherService's error manager.
Alternate Operation	**public String lastErrorStr()** invokes this method but returns the result as a numeric String.

6.4.2.4 Get Error Message

Method	public String getErrorMessage(int error)
Description	Retrieves the text corresponding to the error number provided.
Error settings after call	Service: Not changed Instance: N/A
Special Conditions	See the list of errors in Appendix A. If the value provided is outside the range of valid error numbers, the returned string will contain "Invalid error number."
Usage Comments	Call at any time to obtain the text string describing a particular error number.
Alternate Operation	None.

6.4.2.5 Get Detail Error Message

Method	public String getDetailErrorMessage()
Description	Retrieves the text of the Detail Error Message set by the most recent operation. This operation will return the string set by the most recently called service operation; if none was set, it returns “null”.
Error settings after call	Service: Not changed Instance: N/A
Special Conditions	None.
Usage Comments	Call after determining that an error at the service has occurred to see if a more detailed description of the problem is available.
Alternate Operation	None.

6.4.2.6 Get Exception Message

Method	public String getExceptionMessage()
Description	Retrieves the Exception message text, if one occurred during the most recent operation, or a “null” value.
Error settings after call	Service: Not changed Instance: N/A
Special Conditions	None.
Usage Comments	Call after determining that an error at the service has occurred to see if an exception resulted and get more detailed information.
Alternate Operation	None.

6.4.2.7 Check for Instance Error

Method	public boolean instanceError(String token)
Description	Determine if an error occurred in the most recent command sent to the SIMPROCESS instance identified by the unique token.
Error settings after call	Service: Set to UNKNOWN_INSTANCE if the token is unknown; otherwise NOERR. Instance: Not changed
Special Conditions	This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.
Usage Comments	Call after most instance operations to determine if a non-zero error (any error other than NOERR) exists in a SIMPROCESS instance's error manager. Any time this call returns true, first confirm that service's error number is NOERR and then check instance's value.
Alternate Operation	**public String instanceErrorStr(String token)** invokes this method and returns the result as a String containing "true" or "false".

6.4.2.8 Last Instance Error Number

Method	public int lastInstanceError(String token)
Description	Get last error number for the SIMPROCESS instance identified by the unique token.
Error settings after call	Service: Changed to UNKNOWN_INSTANCE if the token is unknown; otherwise NOERR. Instance: Not changed
Special Conditions	This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.
Usage Comments	Call after most instance operations to obtain the current error number value from an instance's error manager. An invalid token returns UNKNOWN_INSTANCE.
Alternate Operation	**public String lastInstanceErrorStr(String token)** invokes this method and returns the result as a numeric String.

6.4.2.9 Get Instance Error Message

Method	public String getInstanceErrorMessage(String token)
Description	Retrieves the text corresponding to the current error number of the instance identified by the unique token.

Error settings after call	Service: UNKNOWN_INSTANCE if token is unknown; otherwise unchanged. Instance: Not changed
Special Conditions	See the list of errors in Appendix A. This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.
Usage Comments	Call to get the text corresponding to the instance's current error number. Will return "null" if the instance is unknown.
Alternate Operation	None.

6.4.2.10 Get Detail Instance Error Message

Method	public String getDetailInstanceErrorMessage(String token)
Description	Retrieves the text of the Detail Error Message set by the most recent operation of the instance. This operation will return the string set by the most recent operation; if none was set, it returns "null".
Error settings after call	Service: Set to UNKNOWN_INSTANCE if token not known; otherwise not changed. Instance: Not changed
Special Conditions	This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.
Usage Comments	Call after determining that an error occurred in most recent operation of the instance referred to by token; "null" will result if the service does not know the token or if no value has been set by the instance.
Alternate Operation	None.

6.4.2.11 Get Instance Exception Message

Method	public String getInstanceExceptionMessage(String token)
Description	Retrieves the Exception message text, if one occurred during the most recent operation of the instance identified by the unique token, or a "null" value if none occurred.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown; otherwise not changed. Instance: Not changed
Special Conditions	This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.

Usage Comments	Call after determining that an error occurred during the most recent instance operation. If a "null" is returned, verify service did not set UNKNOWN_INSTANCE.
Alternate Operation	None.

6.4.2.12 Get Default Instance Timeout

Method	public int getDefaultTimeout()
Description	Retrieves the length of time in minutes used as the default timeout value for new instances.
Error settings after call	Service: None Instance: None
Special Conditions	None.
Usage Comments	This corresponds to the instance.timeout.default property of the dispatcherservice.properties file.
Alternate Operation	**public String getDefaultTimeoutStr()** invokes this method and returns the value as a numeric String.

6.4.2.13 Get Maximum Instance Timeout

Method	public int getMaxTimeout()
Description	Retrieves the length of time in minutes which is the largest timeout value allowed for any instance. This value will be used any time a timeout value is set which exceeds it.
Error settings after call	Service: None Instance: None
Special Conditions	None.
Usage Comments	Corresponds to the value in the instance.timeout.max property of the dispatcherservice.properties file.
Alternate Operation	**public String getMaxTimeoutStr()** invokes this method and returns the value as a numeric String.

6.4.2.14 Get Instance's Timeout

Method	public int getTimeout(String token)
Description	Retrieves the timeout setting of the SIMPROCESS instance identified by the unique token, or a -1 if any error occurs.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown; otherwise unchanged. Instance: No change

Special Conditions	This call **does not** get passed to the specified instance, and therefore **does not** delay its timeout.
Usage Comments	
Alternate Operation	**public String getTimeoutStr(String token)** invokes this method and returns the value as a numeric String.

6.4.2.15 Set Instance's Timeout

Method	public boolean setTimeout(String token, int newTimeout)
Description	Sets the timeout period of the SIMPROCESS instance identified by the unique token to the number of minutes specified in newTimeout.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown; otherwise unchanged. Instance: NOERR
Special Conditions	This call restarts the instance's timeout countdown.
Usage Comments	If the value of newTimeout is less than zero, the default timeout is used. If larger than the maximum timeout allowed, that value is used. Returns "false" only if the instance token is unknown.
Alternate Operation	**public String setTimeoutStr(String token, String newTimeout)** accepts a numeric string for newTimeout (forced to -1 if not a valid numeric value), invokes this method and returns its result as a String containing "true" or "false".

6.4.2.16 Start SIMPROCESS Instance

Method	public String startInstance(int timeout)
Description	Requests launch of a SIMPROCESS instance with the specified timeout duration; returns a unique token referring to that instance or "null" if unable to start one.
Error settings after call	Service: CANNOT_CREATE_TOKEN if unable to obtain a unique token NO_AVAILABLE_DISPATCHERS if there are no dispatchers to handle the request One of several possible error settings of all available Dispatchers are unable to start an instance Instance: NOERR
Special Conditions	This call restarts the instance's timeout countdown.

Usage Comments	If the value of timeout is less than zero, the default timeout is used. If larger than the maximum timeout allowed, that value is used. Timeout countdown begins on successful start.
Alternate Operation	**public String startInstanceStr(String timeout)** accepts a numeric string for timeout (forced to -1 if not a valid numeric value), then invokes this method and returns the token received.

6.4.2.17 Stop SIMPROCESS Instance

Method	public boolean stopInstance(String token)
Description	Requests shutdown of the SIMPROCESS instance identified by the unique token.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted One of several possible errors returned from the owning Dispatcher if the instance could not stop, including MODEL_RUNNING if instance is currently simulating Instance: Not changed
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	A false response not accompanied by an error setting of MODEL_RUNNING should probably be followed by a forced shutdown request.
Alternate Operation	**public String stopInstanceStr(String token)** invokes this method and returns the result as a String containing "true" or "false".

6.4.2.18 Have SIMPROCESS Instance Say Hello

Method	public String instanceHello(String token)
Description	Ask the SIMPROCESS instance identified by the unique token to "say hello" to confirm communication.

Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: One of several possible errors returned from the instance
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	This operation may be useful as a means of restarting an instance's timeout countdown or merely to confirm the validity of a token. It will result in a null response if any error condition occurs.
Alternate Operation	None

6.4.2.19 Open SIMPROCESS Model File

Method	public boolean openModel(String token, String model)
Description	Ask the SIMPROCESS instance identified by the unique token to open the named model file. The value of the *model* parameter must correspond to a model file in the SIMPROCESS installation's *models* directory on the system where the instance is running; it is case sensitive and must not include the ".spm" extension.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_NOT_FOUND if the named model file is not present in the *models* directory (after adding the extension ".spm"), or other error as appropriate.
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	This operation may be useful as a means of restarting an instance's timeout countdown or merely to confirm the validity of a token. It will result in a null response if any error condition occurs.
Alternate Operation	**public String openModelStr(String token, String model)** invokes this method and returns its result as a String containing "true" or "false".

6.4.2.20 List Open SIMPROCESS Models

Method	public String[] listOpenModels(String token)
Description	Get a list of open models from the instance identified by the unique token. The result will be "null" if an error occurs, or an array of strings with zero or more entries for the names of all open models.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: NO_ACTOR if an internal configuration error has occurred, or other error as appropriate.
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	If a null response occurs, check the service error setting first, then the instance setting.
Alternate Operation	None

6.4.2.21 Close SIMPROCESS Model

Method	public boolean closeModel(String token, String model)
Description	Ask the instance identified by the unique token to close the specified model file.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_ALREADY_RUNNING if specified model currently simulating MODEL_NOT_OPEN if named model is not open Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	Closing a model is not required prior to shutting down an instance.

Alternate Operation	**public String closeModelStr(String token, String model)** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.22 Set Model Parameters

Method	public boolean setModelParameters(String token, String model, String params[])
Description	Set parameters of the named model in the instance identified by the unique token. Valid array contents are the same as what is described when using a properties file in Appendix J of the SIMPROCESS User's Manual under the heading **Available Options**. However, the automatic commit to a database does not occur when Design and Scenario values are set. That behavior is applicable only to the traditional single-user environment of SIMPROCESS.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_ALREADY_RUNNING if specified model currently simulating MODEL_NOT_OPEN if named model is not open PARAMETER_ERROR if the parameter array is null or another error results from invalid or incorrect contents of "param" Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	This operation is useful not only for setting the initial values of Attributes designated as Model Parameters, but also for selecting which alternative sub-process to make the active one for any Processes which have multiples (the currently set one will be used if not specified). If a false response results, check the service's error setting first and then the instance's.
Alternate Operation	**public String setModelParametersStr(String token, String model, String params[])** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.23 Start Simulation

Method	public boolean startSimulation(String token, String model)
Description	Start simulation of the named model in the instance identified by the unique token.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_ALREADY_RUNNING if specified model is already simulating MODEL_NOT_OPEN if named model is not open MODEL_ RUNNING if another model is already simulating (an instance may only simulate one model at any time) INITIALIZATION_ERROR if an error occurred during simulation initialization, such as expression errors Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	This operation starts the simulation process for a model. Once simulation begins, the instance will not honor requests to shut down until it has completed, whether normally, on request, or by error. If a false result occurs, check the service's error and then the instance's.
Alternate Operation	**public String startSimulationStr(String token, String model)** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.24 Stop Simulation

Method	public boolean stopSimulation(String token, String modelName)
Description	Stop simulation of the named model in the instance identified by the unique token.

Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_NOT_OPEN if named model is not open MODEL_ NOT_RUNNING if named model is not simulating SIMULATION_STOP_FAILURE if an error caused the instance to be unable to stop the simulation (the instance will wait a maximum of 15 seconds for successful termination; a forced instance shutdown may then be the only means of terminating the simulation if this error occurs) Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	This operation stops the simulation process for a model. Once simulation stops, the instance will still honor requests for information on the simulation status until the model is closed. When a false result occurs, check the service's error and then the instance's error.
Alternate Operation	**public String stopSimulationStr(String token, String model)** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.25 Force SIMPROCESS Instance Shutdown

Method	public boolean forceInstanceDown(String token)
Description	Forcefully shut down the instance identified by the unique token.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Other error if appropriate Instance: N/A
Special Conditions	

Usage Comments	This operation instructs the owning Dispatcher to terminate an instance without regard to any ongoing operations, such as opening or simulating a model. Depending on numerous conditions, such operations can be lengthy, and this operation should not be used lightly. In the event of a "false" reply when it is certain that no failure of the DispatcherService, RegistryServer, or Dispatcher has occurred, it may be necessary to have a system administrator determine whether a process is in need of special action to terminate.
Alternate Operation	**public String forceInstanceDownStr(String token)** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.26 Is Simulation Complete?

Method	public boolean isSimulationComplete(String token, String modelName)
Description	Inquire whether simulation of the named model in the instance identified by the unique token has completed (normally or otherwise).
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_NOT_OPEN if named model is not open MODEL_ NOT_SIMULATED if named model has not simulated since opening Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	A simulation should be considered complete only when a "true" reply is received along with both the service and instance error settings of NOERR. This does not indicate a normal completion, however, only that the simulation has been completed. Additional simulation status should be checked to determine if completion was normal.
Alternate Operation	**public String isSimulationCompleteStr (String token, String model)** invokes this operation and returns its result as a String containing "true" or "false".

6.4.2.27 Get Replication Number

Method	public int getReplication(String token, String modelName)
Description	Get the current replication number for the simulation of the named model in the instance identified by the unique token. A value of -1 will indicate an error; see below.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_NOT_OPEN if named model is not open MODEL_ NOT_SIMULATED if named model has not simulated since opening Other error if appropriate
Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	
Alternate Operation	**public String getReplicationStr(String token, String modelName)** invokes this operation and returns its result as a numeric String.

6.4.2.28 Get Simulation Status

Method	public SimulationStatus getSimulationStatus(String token, String modelName)
Description	Retrieves a SimulationStatus object with information about the simulation of the named model in the instance identified by the unique token. See Appendix B for a Java source listing of SimulationStatus.
Error settings after call	Service: UNKNOWN_INSTANCE if the token is unknown INSTANCE_BUSY if another operation is currently being performed by the instance DISPATCHER_UNAVAILABLE if the owning Dispatcher cannot be contacted Instance: MODEL_NOT_OPEN if named model is not open MODEL_ NOT_SIMULATED if named model has not simulated since opening Other error if appropriate

Special Conditions	Restarts the instance's timeout countdown.
Usage Comments	SimulationStatus consolidates bits of information which are not otherwise available via DispatcherService into a single location, making it a convenient way of gathering them in one call.
Alternate Operation	None

7 Client Applications

DispatcherService was developed entirely in Java using Sun's JAX-RPC and JAXP technologies. As a result, all messages sent to or received from DispatcherService, once it is deployed as a Web service, will be SOAP XML messages. That means the resulting service can be called upon by a client application written in any language and running on any platform, provided it is able to send and receive messages to a Web service using SOAP protocols. One common example might include Java Server Pages (JSPs), which can run in the same Web Container as the DispatcherService itself. The testing done during development used Java clients of varying sophistication. And the new SOAPCall expression statement in SIMPROCESS itself was exercised against DispatcherService.

Appendix A

Error Numbers and Descriptions

Error Constant	Error Number	Error String
NOERR	0	No error
UNKNOWN_ERROR	1	An unknown error has occurred; see detail information.
BAD_CONTEXT	2	Null Context; registry may be unavailable and service should be restarted.
NO_LICENSE_AVAILABLE	3	Unable to obtain a license for a new SIMPROCESS instance.
LAUNCH_FAILED	4	Launch of SIMPROCESS instance failed; see detail information.
INSTANCE_LIST_FAILURE	5	An error occurred retrieving the instance list.
UNAUTHORIZED_DISPATCHER	6	Unauthorized DispatcherServer requesting instance shutdown; ignored.
UNKNOWN_INSTANCE	7	Instance for specified token not found.
LOOKUP_FAILURE	8	DispatcherRegistry lookup error.
UNBIND_FAILURE	9	DispatcherRegistry unbind error; unbind may not have been successful.
REGISTRY_UNAVAILABLE	10	RegistryServer unavailable.
DISPATCHER_UNAVAILABLE	11	Named DispatcherServer unavailable.
LIST_UNAVAILABLE	12	Requested list unavailable; possible service error.
NO_ACTOR	13	Instance has not provided Actor for callback.
FILE_ERROR	14	An I/O error has occurred on a file operation.
FILE_NOT_FOUND	15	Specified model file not found.
MODEL_RUNNING	16	A model is currently simulating.
MODEL_ALREADY_OPEN	17	Requested model file is already open.
MODEL_ALREADY_RUNNING	18	Requested model is currently simulating.
MODEL_OPEN_FAILED	19	An error occurred opening the requested model file; it may be damaged.
MODEL_NOT_OPEN	20	Requested model is not open.
MODEL_NOT_RUNNING	21	Requested model is not simulating.
PARAMETER_ERROR	22	Error setting model parameters; see

Error Constant	Error Number	Error String
		detail information.
INITIALIZATION_ERROR	23	Error during simulation initialization; see detail information.
SIMULATION_STOP_FAILURE	24	Unable to stop simulation; see detail information.
NO_AVAILABLE_DISPATCHERS	25	There are no available Dispatchers.
CANNOT_CREATE_TOKEN	26	Unable to create unique instance ownership token.
MODEL_NOT_SIMULATED	27	Requested model has not run a simulation.
DESIGN_AND_SCENARIO_NEEDED	28	Commit requires both Design and Scenario names.
COMMIT_FAILED	29	Commit unsuccessful; see detail information for further information.
INSTANCE_BUSY	30	Instance busy executing another command.

Notes on Error Numbers and Messages:

1. The presence of any error text relating to a particular function or behavior does not imply that the function is, or will be, available in SIMPROCESS when operated as a service via the DispatcherService and Dispatcher applications.
2. Some error numbers represent special conditions which are theoretically possible but have not actually been encountered during testing.
3. Though some error messages make reference to detail information and others do not, this additional information should be checked following most operations to ensure both completeness and accuracy.

Appendix B

SimulationStatus Source Listing

```
package com.simprocess.service;

public class SimulationStatus {

private String errorMessage = null;
private String exceptionMessage = null;
private boolean error = false;
private boolean simulating = false;
private boolean simComplete = false;
private boolean initializing = false;
private boolean haltedEarly = false;

public SimulationStatus() {
}

public SimulationStatus(String errorMessage, String exceptionMessage,
                        boolean error,
                        boolean simulating,
                        boolean simComplete,
                        boolean initializing,
                        boolean haltedEarly) {
   this.errorMessage = errorMessage;
   this.exceptionMessage = exceptionMessage;
   this.error = error;
   this.simulating = simulating;
   this.simComplete = simComplete;
   this.initializing = initializing;
   this.haltedEarly = haltedEarly;
}

public void setErrorMessage(String msg) {
   this.errorMessage = msg;
}

public String getErrorMessage() {
   return this.errorMessage;
}

public void setExceptionMessage(String msg) {
   this.exceptionMessage = msg;
}

public String getExceptionMessage() {
   return this.exceptionMessage;
}
```

```
public void setError(boolean flag) {
   this.error = flag;
}

public boolean isError() {
   return this.error;
}

public void setSimulating(boolean flag) {
   this.simulating = flag;
}

public boolean isSimulating() {
   return this.simulating;
}

public void setSimComplete(boolean flag) {
   this.simComplete = flag;
}

public boolean isSimComplete() {
   return this.simComplete;
}

public void setInitializing(boolean flag) {
   this.initializing = flag;
}

public boolean isInitializing() {
   return this.initializing;
}

public void setHaltedEarly(boolean flag) {
   this.haltedEarly = flag;
}

public boolean isHaltedEarly() {
   return this.haltedEarly;
}

}
```