OpenEdge™ Revealed: Achieving Server Control with Fathom™ Management

Doug Merrett

Expert Series

August 2004

Product Code: 4010

Release: V3.0A

Item Number: 101212

Acknowledgements

I'd like to thank all my colleagues who have reviewed this book and given valuable comments to add clarity to the text and also to point out where I had missed important information.

My thanks also go to the Fathom Management development, quality assurance, and documentation teams who have worked hard to provide me with invaluable assistance throughout the writing of this book.

Contents

Figures

Tables

Examples

Preface

This Preface contains the following sections:

Purpose

This manual documents the best practices for configuring the deployment, maintenance, tuning, and debugging of your Progress® OpenEdge™ server-based applications.

Additionally, the Fathom™ Management features that can help you maintain, tune, and monitor the OpenEdge servers are identified at the appropriate times throughout this manual.

Audience

This manual is mainly for use by OpenEdge system administrators. Some parts of this manual pertain to the development and debugging of applications based on the OpenEdge server technology. These will be of interest to Development and Quality Assurance staff.

Fathom Management with OpenEdge or Progress

Fathom Management Version 3.0A runs against the following:

- OpenEdge 10.0B.
- Progress Version 9.1D and the 9.1D09 service pack.

For the sake of simplicity, the procedures and screen shots provided in this manual refer to running Fathom against OpenEdge 10.0B. However, be assured that unless indicated otherwise, the procedures are the same for both Progress Version 9.1D with the 9.1D09 service pack and OpenEdge 10.0B. For example, if a procedure refers to an OpenEdge database, the procedure applies to a Progress database as well.

How to use this manual

Read Chapter 1, “Fathom and OpenEdge Servers Overview,” first because it defines and describes the components that make up an OpenEdge server deployment. Then read the sections of this manual that pertain to your deployment environment.

The goal of this book is not to reiterate what is already documented in the Fathom Management documentation set, but rather to explain how you can best use Fathom Management to help manage, maintain, and monitor an OpenEdge application. This manual contains information gathered from the experience of helping to develop, debug, and deploy OpenEdge applications. If Fathom Management had been available when some of these deployments were done, their successful implementation would have been much easier.

What will not be covered in this manual

The following topics will not be covered in this manual:

- Development configuration
- Other server products
- The AppServer Internet Adapter and Secure AppServer Internet Adapter
- Open clients
- WebServices Toolkit (WSTK)

Development configuration

This manual is designed to cover the best practices for the deployment and continued running of Progress OpenEdge server-based applications. Even though we will cover some debugging and tuning of these applications, the configuration of the development and test servers will not be covered. Some of the deployment information will be useful for the configuration of a “live” test environment.

Other server products

The OpenEdge database server, DataServers, and the SonicMQ® 4GL Adapter can all be managed by the AdminServer. These are not covered in this manual. The database server is covered in *OpenEdge Revealed: Mastering the OpenEdge Database with Fathom Management*, and the others are not currently managed or monitored by Fathom Management.

The AppServer Internet Adapter and Secure AppServer Internet Adapter

The AppServer™ Internet Adapter (AIA) and Secure AppServer™ Internet Adapter (AIA/S) are designed to allow AppServer clients (either 4GL or open clients) to connect to the AppServer over HTTP, or in the case of AIA/S over HTTP/S. This is done to provide ease of deployment for extranet solutions where the client might be behind a firewall and cannot communicate to the AppServer using the normal process. Both adapters are described in *OpenEdge Application Server: Administration*.

The AIA and AIA/S work in a very similar fashion to the WebSpeed Messenger in that they:

- Are accessed via a Web server.
- Take a request from an external client.
- Find the appropriate AppServer broker and AppServer server to send the request to.
- Pass the request through to the AppServer server.
- Send the resulting response to the external client.

The AIA and AIA/S are Java ™ Servlets and run in a Java Servlet Engine (JSE). The JSE can be either on the same machine as the Web server or another machine.

Connecting from the AIA or AIA/S to the AppServer follows the same process as the WebSpeed Messenger, so any information in this manual regarding firewall configuration to allow communications between the WebSpeed Messenger and WebSpeed® Transaction Server is equally applicable to the AIA and AIA/S accessing the AppServer.

Open clients

Open clients are either Java or COM objects that expose an AppServer process to their parent language. Open Clients allow Visual Basic or your Web-enabled applications written in Java to work with the same business functions used by your OpenEdge enterprise applications. The process for connecting to the AppServer is the same as for a 4GL client, so the configuration section of this manual describing 4GL clients connecting to an AppServer is relevant. Open Clients can also access the AppServer via the AIA or AIA/S. The Open Client is fully described in *OpenEdge Development: Open Client Introduction and Programming*.

WebServices Toolkit (WSTK)

The WebServices Toolkit is a product that exposes an AppServer process as a WebService as defined by the World Wide Web Consortium (`http://www.w3c.org`).

For more information on the WSTK, please go to the Progress Software Developers Network (PSDN) Web site (`http://psdn.progress.com`) and look for "WebServices Toolkit" in the PSDN library's "Product and Technology Information" section.

Organization

This manual contains the following chapters:

Chapter 1, "Fathom and OpenEdge Servers Overview"

Provides an overview of the OpenEdge servers, including the OpenEdge Architecture and how you can integrate Fathom Management with your OpenEdge servers.

Chapter 2, "Setup and Configuration of OpenEdge Servers"

Provides general setup and configuration information that are common to each type of OpenEdge server, including WebSpeed®, AppServer™, and NameServer. Topics discussed include broker ownership, the request round-trip process, firewall configuration and debugging, and IP issues.

Chapter 3, "Configuration of WebSpeed Servers"

Provides setup and configuration information that is specific to WebSpeed. Topics discussed include general WebSpeed server configuration, installation of the WebSpeed Messenger, security, error handling and debugging, using the AppServer to access business logic, and simple and complex configurations.

Chapter 4, “Configuration of the AppServer”

Provides setup and configuration information that is specific to the AppServer. Topics discussed include general AppServer configuration, security, error handling and debugging.

Chapter 5, “Configuration of the NameServer”

Provides setup and configuration information that is specific to the NameServer. Topics discussed include an overview of the NameServer, location independence, load-balancing, fault-tolerant NameServer configurations, and NameServer neighborhoods.

Chapter 6, “Performance Considerations”

Provides performance considerations and tuning for WebSpeed, the AppServer, and the NameServer. It also provides performance tuning information for your applications.

Chapter 7, “Where to Use Fathom Management when Deploying”

Provides guidelines for using several Fathom Management features with OpenEdge servers. Topics include using Fathom Management to monitor the AppServer and WebSpeed; debugging the NameServer; monitoring memory; CPU and disks; managing log files; and using My Fathom.

Typographical conventions

This manual uses the following typographical conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, provides emphasis, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge™ key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1-KEY2	A hyphen between key names indicates a *simultaneous* key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL-X**.

Convention	Description
KEY1 KEY2	A space between key names indicates a *sequential* key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax statements, code examples, and for system output and filenames.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax statements.
`Fixed-width bold`	Fixed-width bold indicates variables with special emphasis.
`UPPERCASE fixed width`	Uppercase words are Progress® 4GL language keywords. Although these always are shown in uppercase, you can type them in either uppercase or lowercase in a procedure.
	This icon (three arrows) introduces a multi-step procedure.
	This icon (one arrow) introduces a single-step procedure.

1

Fathom and OpenEdge Servers Overview

This chapter introduces you to Fathom™ and OpenEdge™ servers. It covers the following topics:

- Architecture of OpenEdge servers
- Fathom integration with OpenEdge servers

Architecture of OpenEdge servers

The OpenEdge™ servers are the WebSpeed® Transaction Server and the AppServer™. The NameServer is a separate (and optional in Progress® Version 9.1D and higher) server that is used by the OpenEdge Server environment. The OpenEdge and NameServers are controlled and configured by the Administration server.

Figure 1–1 shows the general architecture of a Progress OpenEdge application deployment.

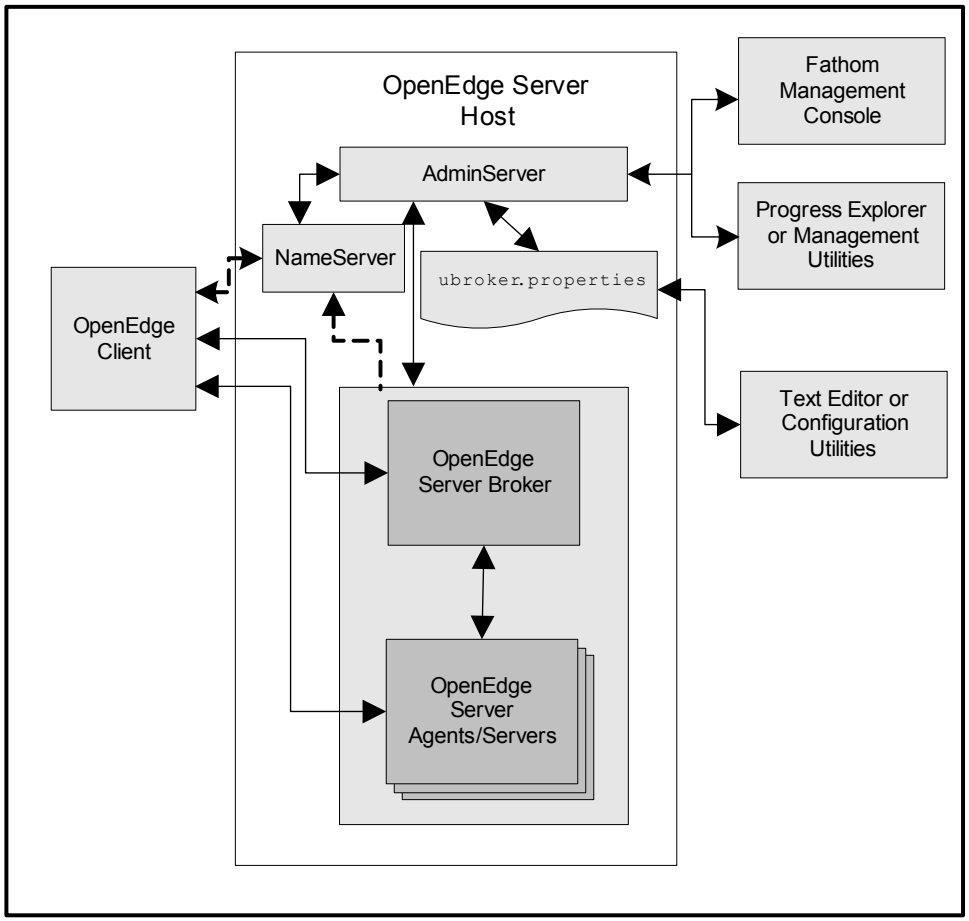


Figure 1–1: Progress OpenEdge application deployment architecture

The Administration server (AdminServer) is the controlling process for all of the OpenEdge servers (including databases, NameServers, DataServers, and so on) on a physical host. The AdminServer gets configuration information for the OpenEdge servers from the `ubroker.properties` file. You can maintain this file using either the Progress Explorer tool or a text editor. Using the Progress Explorer tool is the recommended way to update the `ubroker.properties` file because it validates your input, which helps to alleviate configuration errors.

Figure 1–1 shows only OpenEdge servers and a NameServer, which are all on the same host. However, you can install OpenEdge servers and NameServers on different hosts, and in this case, each host would need its own AdminServer. The solid lines show communication paths and the dashed line indicates optional communication.

The OpenEdge server can be either a WebSpeed server or an AppServer. In the case of WebSpeed, the WebSpeed broker will manage WebSpeed agents. For an AppServer, the AppServer broker will manage AppServer servers.

The NameServer does the following:

- Allows location independence for OpenEdge servers.
- Provides fault tolerance.
- Provides load balancing for requests.

See Chapter 5, “Configuration of the NameServer,” for more information.

Administration server

The AdminServer is the controlling process for all of the OpenEdge servers on a host. It allows the Progress Explorer, command line management utilities, or Fathom Management to start, stop, or query the status of any OpenEdge server.

Note: Fathom Management can control only OpenEdge databases, AppServers, WebSpeed servers, and NameServers.

Starting the AdminServer

Because the AdminServer is the controlling process, it must be started first. On Microsoft Windows, this is automatically achieved because the AdminServer is installed as a Service.

On UNIX, if you want the AdminServer to automatically start, add the `proadsv -start` command to a script in the `rc.d` directories. Please consult your UNIX administration guide for details.

Understanding the ubroker.properties file

If you look at the `ubroker.properties` file (located the *`OpenEdge-install`*`/properties` directory) using a text editor, you will see each OpenEdge server that can run on that host.

Example 1–1 shows an example of a WebSpeed server configuration in the `ubroker.properties` file.

Example 1–1: ubroker.properties file

```
[UBroker.WS.DemoApp]
    srvrAppMode=Production
    brokerLogFile=C:\DemoApp\Logs\WS_broker.log
    srvrLogFile=C:\DemoApp\Logs\WS_agent.log
    srvrMinPort=5321
    srvrMaxPort=5325
    maxSrvrInstance=5
    controllingNameServer=NS1
    srvrStartupParam=-p web\objects\web-disp.p -weblogerror
    uuid=527a0623fe008210:67d940:f6484c1312:-7d87
    workDir=C:\DemoApp
```

Each broker has a Universally Unique Identifier (UUID). The UUID must be unique across your entire configuration in order for the NameServer to function properly. If you want to manually add a new OpenEdge server to the `ubroker.properties` file, use the `genuuid` script to generate a UUID. The `genuuid` script is located in the `DLC/bin` directory.

For complete details on the `ubroker.properties` file, see *OpenEdge Getting Started: Installation and Configuration for Unix* or *OpenEdge Getting Started: Installation and Configuration for Windows.*

WebSpeed

WebSpeed® is a development and deployment environment that allows you to build applications that use HTML, XML, WML, DHTML, and most other mark-up languages (MLs) as the user interface. This means that WebSpeed can be used for applications where users are accessing the application using:

- A Web browser (HTML, DHTML, or XML).
- A mobile/cell-phone (HTML, WML).
- A system making requests for information using XML and HTTP or HTTP/S as the transport protocol.

The WebSpeed language is the Progress 4GL with some extensions. So, generally speaking, anything that Progress 4GL can do, WebSpeed can do. There are three different development techniques for WebSpeed applications:

- Embedded SpeedScript
- Mapped Web objects
- CGI wrappers

Embedded SpeedScript is a technique whereby Progress 4GL is inserted into HTML by using special HTML tags. When the WebSpeed compiler compiles this HTML, it converts it into a CGI wrapper behind the scenes.

Mapped Web objects and CGI wrappers are pure 4GL applications, and the HTML is treated as "text" within the program. For a complete description of the WebSpeed development environment and techniques, see *OpenEdge Application Server: Developing WebSpeed Applications.*

WebSpeed server

The WebSpeed server is made up of two components: the WebSpeed broker and the WebSpeed agents. A third component, called the WebSpeed Messenger, is illustrated in the WebSpeed architecture diagram, as shown in Figure 1–2.

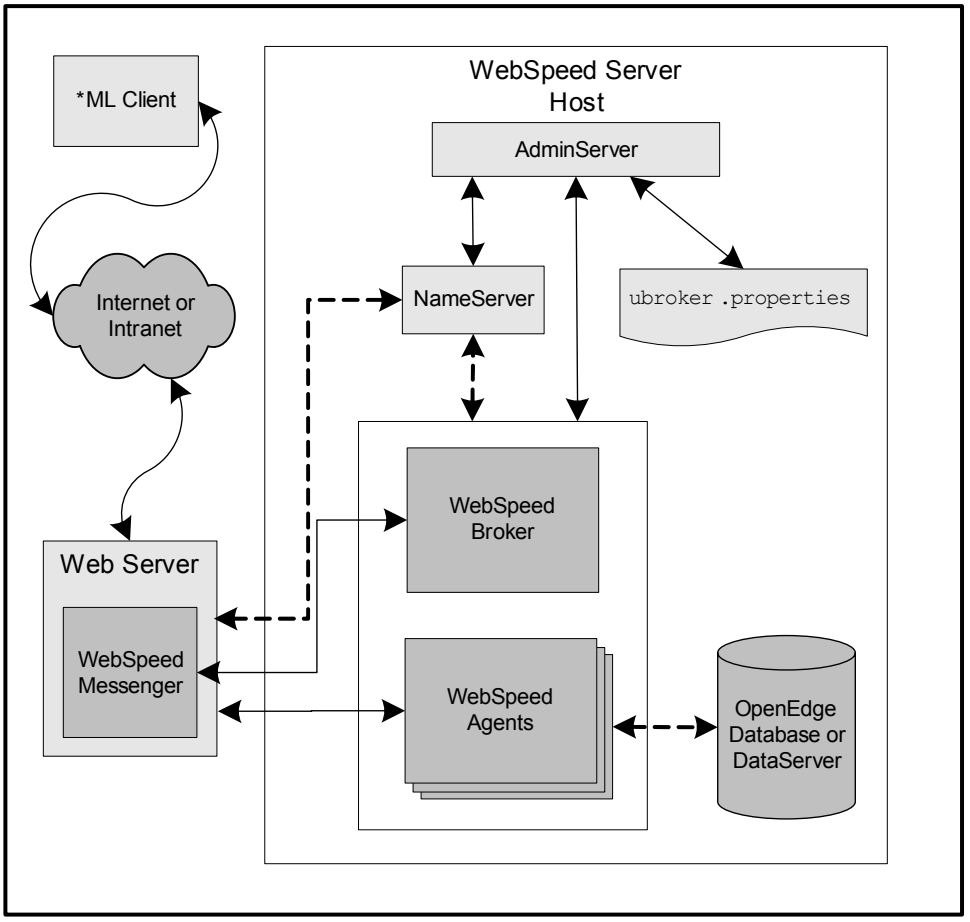


Figure 1–2: WebSpeed deployment architecture

Figure 1–2 shows the complete architecture for a WebSpeed deployment environment. The NameServer is optional, as described in the "No NameServer version of the request round-trip" section on page 2–8, and if it is used, it may be on another host. The WebSpeed agents are shown connecting to an OpenEdge database or DataServer. This is optional because the WebSpeed agents can connect to an AppServer to process the business logic. The database or DataServer, if used, can also be on another host.

WebSpeed broker and agents

The WebSpeed broker manages the WebSpeed agents. When the WebSpeed server is started by the AdminServer, the WebSpeed broker is started first. Then the AdminServer passes configuration information to the broker including how many agents to start initially, agent startup parameters, agent port ranges, and other settings. After the agents are started, the broker tells the specified NameServer that it is "alive" and able to take requests for the specified service.

During the lifetime of the WebSpeed server, the broker:

- Tells the Messenger which agent is available to handle incoming requests.
- Spawns more agents if needed (up to the specified limit).
- Trims (or kills) agents if they have not been used for the specified number of seconds.
- Responds to queries from the AdminServer for performance metrics.
- Sends the AdminServer information if an event occurs, like starting or trimming agents.

The WebSpeed agents are 4GL clients that will accept requests from the WebSpeed Messenger and run the requested program. The output of the program is sent through a special stream to the Messenger and then through the Web server to the requesting client.

The agent process is inherently stateless. This means that the agent is only busy when a request is being processed. It will be idle at all other times.

AppServer

The AppServer is a deployment environment that allows Progress 4GL clients to run Progress 4GL business logic on a different machine. Examples of 4GL clients include: GUI, Character, Batch, WebSpeed, and other AppServers, or via the OpenClient Toolkit, Java or COM clients.

There are two main benefits to having an AppServer in the deployment configuration of an application:

1. The user interface and the business logic are separated, which allows different user interfaces (such as GUI, Batch, and/or WebSpeed) to share the same business logic.
2. The Progress 4GL application becomes a thin client application—the network traffic between the client and database is reduced because the only information travelling along the network is the request and reply.

For example, if you used a query that was not indexed, the client would need to download all the table's information and search through it locally. Whereas in an AppServer version, the client would send the request to the AppServer, which would access the database locally and send only the matching records back to the client, thereby reducing the network traffic.

The AppServer is made up of two components: the AppServer broker and the AppServer servers, as shown in Figure 1–3.

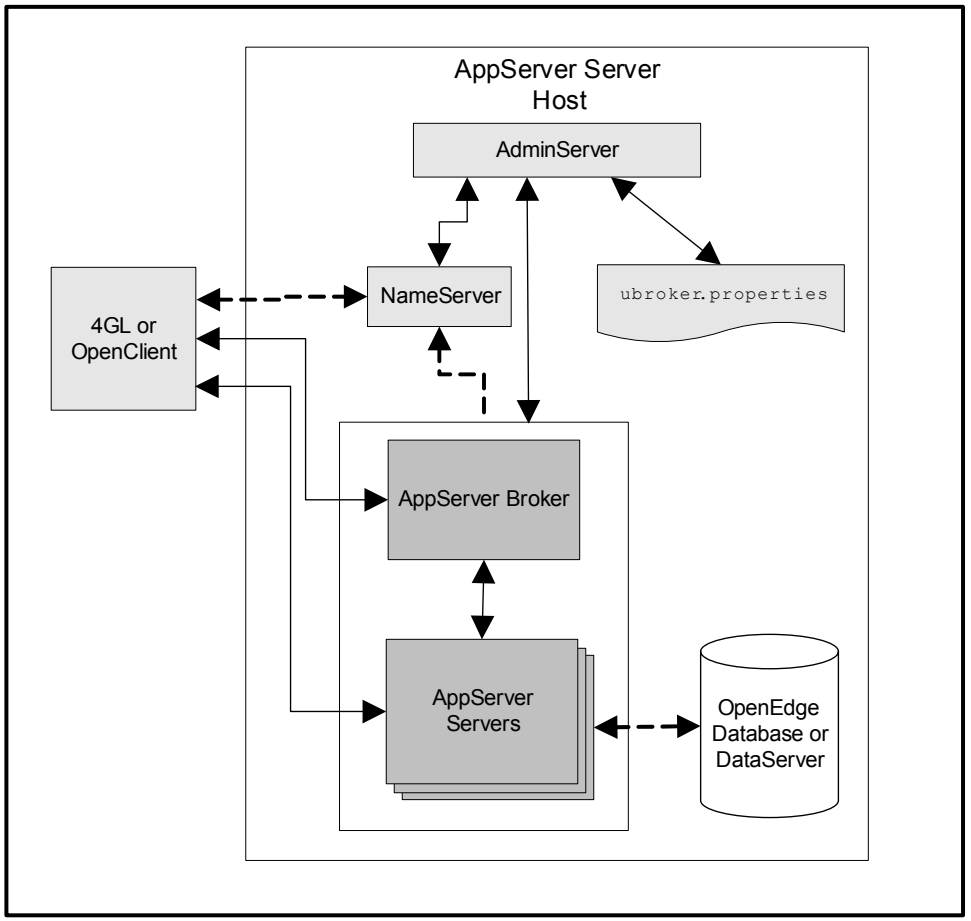


Figure 1–3: AppServer deployment architecture

Figure 1–3 shows the complete architecture for an AppServer deployment environment. The NameServer is optional (as discussed in the "Understanding the NameServer" section on page 5–2) and if it is used, it may be on another host. The AppServers are shown connecting to a OpenEdge database or DataServer. This is optional because the servers may connect to another AppServer to process the business logic, making an n-tier application. The database or DataServer, if used, may also be on another host. As a general rule, for best performance, the AppServer and the database/DataServer should be on the same host.

AppServer broker and servers

The AppServer broker performs exactly the same processes as the WebSpeed broker described in the "WebSpeed broker and agents" section on page 1–7, but for AppServer servers, not WebSpeed agents.

The AppServer servers are 4GL clients that accept a `RUN` request from other 4GL (or Open) clients. Normally, the client and AppServer are on different machines. This `RUN` command is syntactically the same as a normal `RUN`, and can accept both input and output parameters. The difference is that the `RUN` command specifies which AppServer to run the procedure on via a server handle. For specific details, see *OpenEdge Application Server: Developing AppServer Applications*.

You can configure the AppServer broker to be in one of the following operating modes: state-reset, state-aware, or stateless. These are documented in *OpenEdge Application Server: Developing AppServer Applications*.

Fathom integration with OpenEdge servers

The AdminServer uses server plugins to integrate the OpenEdge components into the AdminServer framework. Fathom is also integrated into this framework using the same method. This allows the Fathom server plugin to interrogate the OpenEdge server plugins to gather information.

Fathom can start, stop, and query the status of the OpenEdge servers as well as display the broker and server log files. These features are available in the Progress Explorer.

Fathom cannot yet be used to configure the OpenEdge servers, so you will still need to use the Progress Explorer or edit the `ubroker.properties` file manually. The ability to use Fathom as a Progress Explorer replacement is planned for a future release.

The Progress Explorer cannot access the detail of information that Fathom uses to provide detailed performance statistics from OpenEdge servers. These performance statistics are one of the most interesting and useful parts of Fathom when used to help monitor deployed OpenEdge applications.

2

Setup and Configuration of OpenEdge Servers

This chapter discusses general configuration tasks that are common to OpenEdge servers, and contains the following topics:

- General OpenEdge server configuration tasks
- Broker ownership
- Request round-trip process
- Firewall configuration and debugging
- IP issues

For additional configuration tasks and information specific to each type of OpenEdge server, see:

- Chapter 3, “Configuration of WebSpeed Servers”
- Chapter 4, “Configuration of the AppServer”
- Chapter 5, “Configuration of the NameServer”

General OpenEdge server configuration tasks

For a deployment host, you should use AdminServer security. When you are installing OpenEdge onto a Windows host, the installation process will prompt you for "AdminServer Authorization Options." If you are installing onto a UNIX host, you apply the AdminServer security as a post installation procedure. These processes are fully described in *OpenEdge Getting Started: Installation and Configuration for Unix* and *OpenEdge Getting Started: Installation and Configuration for Windows*.

When you install OpenEdge onto a server, you will get a default `ubroker.properties` file. This file contains an example configuration for each server (even if you do not have a license for that server).

In a development and test environment, it is safe to use these default server configurations. In a deployment environment, do not use any standard OpenEdge server configurations. This is because they use "well-known" ports and are more likely to be left in their default (which is unsafe) mode.

In the `ubroker.properties` file, you should delete the `asbroker1`, `wsbroker1`, `ns1`, `mssbroker1`, and the others. This can be done through the Progress Explorer or using any text editor.

When you create the new OpenEdge servers, you should create the NameServer first, and then the OpenEdge servers so you can set their NameServer to the one you have just created.

Do not use the standard port numbers for the brokers and servers/agents. For example, when creating a WebSpeed Transaction Server with a maximum of five agents:

- The broker uses port 5320.
- The agents will use ports 5321 to 5325, inclusive.
- If you want more agents, just increase the agent port range upper value.

Later, when you need to configure access to this WebSpeed Transaction Server through a firewall it will be easier if you keep the port range as small as possible. You should not use ports that are below 1024 because these are deemed to be system ports and generally have higher access to the kernel. It is good practice to document the ports you use in the networking "services" file. Under UNIX, this is `/etc/services` and in Windows this is `C:\WINDOWS\system32\drivers\etc\services`. Progress Explorer will not let you use the service name, only the number. If you use a text editor to modify the `ubroker.properties`, you must also use the number, not the service name.

Note that there is a bug in Windows that will not let the last line of the services file (or any other text file) to be read unless it has a carriage-return/line-feed at the end. You might want to add a line, such as the following, to the end of the services file to ensure that there is a carriage-return/line-feed at the end of each "real" entry in the services file:

```
# Do not delete this line
```

If you do not have a carriage-return/line-feed at the end of the services file, the broker process will read it incorrectly and you will get errors, such as the following:

```
Unable to find server XXX with protocol TCP in file SERVICES or SERVICES file
not found in expected location. (5192).
```

To change the port used by the AdminServer under Windows:

1. In the Windows Control Panel Services, stop the AdminServer service.
2. Run `REGEDIT`. You must add the `-port` argument along with the port number to the end of the startup and shutdown entries in your registry. The registry keys that need to be modified are:
   - `HKEY_LOCAL_MACHINE\SOFTWARE\PSC\AdminService\`*`OpenEdgeVersion`*`\StartupCmd`
   - `HKEY_LOCAL_MACHINE\SOFTWARE\PSC\AdminService\`*`OpenEdgeVersion`*`\ShutdownCmd`

The following example shows a changed StartupCmd:

```
"C:\DLC91D\bin\jvmstart" -o eventmgr -w @{WorkPath} @{JAVA\JREHOME}\bin\java
-classpath @{JAVA\JRECP};@{JAVA\PROGRESSCP};@{JAVA\FATHOMCP};@{JAVA\REPLCP}
@{JAVA\JVMARGS} -DInstall.Dir=@{Startup\DLC} -DWork.Dir=@{WorkPath}
-Djava.jvmargs=@{JAVA\JVMARGS} -Djava.security.policy=@{JAVA\JAVAPOLICY}
-Dadmsrv.jvm=@{JAVA\JREHOME}\bin\java -Djvmstart.debug=0
com.progress.chimera.adminserver.AdminServerType -start -service -port 20555
```

Be careful when you edit the registry manually. You should not change the other parameters in this registry key, or your AdminServer might not start up again.

Under UNIX, use the following command to start the AdminServer where 20555 is the new port number:

```
proadsv -start -port 20555
```

From this point on, you will need to use this new port when connecting to the AdminServer from either the Progress Explorer or the Management Utilities.

Broker ownership

For the AdminServer to work it needs to be run with root or system administrator rights. When the AdminServer starts an AppServer or WebSpeed server, by default, it also runs as the root/system administrator. This could allow the AppServer or WebSpeed agent to have access to sensitive data. You can limit what the AppServer or WebSpeed agent can access by adding ownership information to the broker. The owner specified can then have operating system privileges reduced to only what is necessary. Be sure to test this thoroughly, as not having correct permissions can cause many problems.

Figure 2–1 shows a WebSpeed broker under Windows being allocated to run as a regular user. This user needs to have certain permissions when running under Windows. Search the Knowledge Center for “How to start a Broker using Domain\Account.” You can access the Progress Knowledge Center by going to the following URL: http://www.progress.com/support/kb.

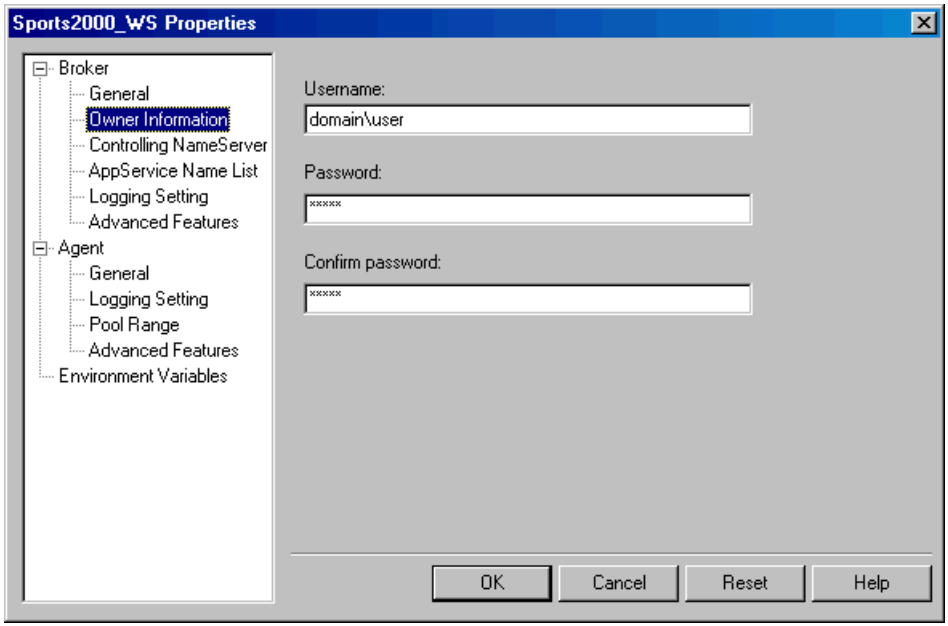


Figure 2–1: Setting broker ownership

When using UNIX or Linux, the fields on this tab are different. They are **Username** and **Group Name**. Set the **Username** and **Group Name** to that of a valid user with appropriate permissions.

Request round-trip process

To connect to either a WebSpeed agent or an AppServer, the client process goes through several steps. The client process for WebSpeed is the WebSpeed Messenger and a 4GL client for the AppServer. These connection steps are shown in Figure 2–2.

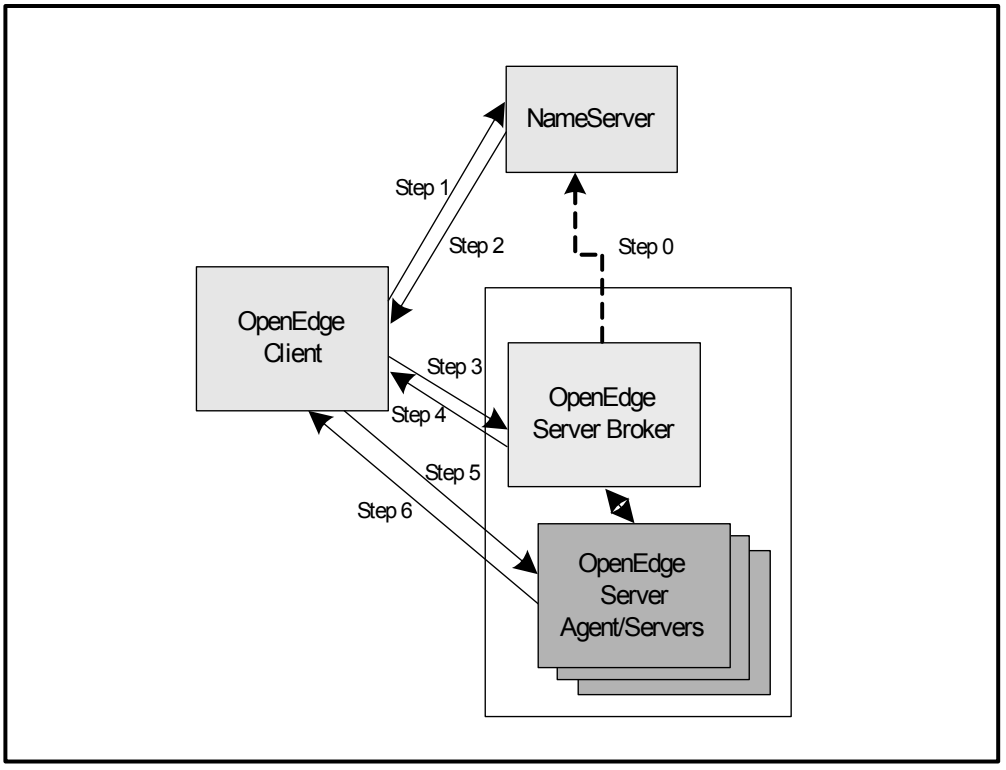


Figure 2–2: Request round-trip

Step 0

When the broker finishes initializing, it tells the NameServer that it is ready to accept requests for specific services and/or be the default service. By default, the broker sends these messages every 30 seconds. This lets the NameServer know the broker is still alive and able to accept requests. If the NameServer does not get a message, it will delete this broker from its "available" list. The broker/NameServer communications are not part of the request process, but are required for it to work. The NameServer is covered in more detail in Chapter 5, "Configuration of the NameServer."

Step 1

The client makes a request to the NameServer for the location of a broker that can provide a "service."

Step 2

The NameServer responds with the broker's host name or IP address and the broker's port number.

Step 3

Using these details, the client connects to the broker and requests a server/agent to be made available. This request is put in a queue by the broker, so in peak load times requests are not lost. If there are requests in the queue, the broker will check to see if there is an available server/agent. If there are none free, and the maximum number of servers/agents has not been reached, the broker will start a new one. When there is a free server/agent, the broker will allocate this one to the request and mark it as busy.

Step 4

The broker returns the server/agent port number or an error if there were none available.

Step 5

Assuming there is a free agent/server, the client uses the port number returned to make a connection to the agent/server and passes the application request.

Step 6

The agent/server responds with the results of the request.

If the OpenEdge server is WebSpeed, then the request is complete and the agent informs the broker that it is again free.

In an AppServer environment, it depends on what "state" the AppServer is configured as. In state-aware or state-reset modes, the AppServer is "tied" to that particular client and can process more requests without the client following Step 1 through Step 4. When the client is totally finished, it will disconnect from the AppServer by using the `DISCONNECT` method. The AppServer will then become free and inform the broker.

In stateless mode, when the client runs the CONNECT method, only Step 1 and Step 2 are followed. The client then connects to the broker and is allocated a unique CONNECTION-ID. The broker does not allocate an AppServer until the client executes a RUN ON SERVER command. Then the client follows Step 3, sending back the CONNECTION-ID, and Step 4 through Step 6 as normal. After the request is completed, the AppServer Server will inform the broker it is free. This means that the client can be allocated different AppServer servers for each request, but only from the same AppServer broker.

The different modes of the AppServer are detailed in *OpenEdge Application Server: Developing AppServer Applications*; their effects on performance are covered in the "How AppServer operating modes affect performance" section on page 6–6 in this manual.

Step 0 through Step 4 create quite small amounts of network traffic, usually less than 500 bytes. The large amounts of data are in the final request and response, Step 5 and Step 6. In an AppServer environment, the request contains all the input parameters, and this can get quite large if there are temp tables being passed as input parameters. When using WebSpeed, the data sent from the Messenger to the WebSpeed agent includes all of the environment variables, as well as the input parameters from the URL or HTML form. The environment variables alone can be up to 3000 bytes. The response for an AppServer call is the entire amount of data returned. This could be as simple as an integer, or as complex as multiple temp tables. All this data can become quite large. When the response comes back from a WebSpeed agent, it could be a simple HTML page of around 1000 bytes, but it also could be a large .ZIP file or similar. With special programming, WebSpeed can send binary files to the Web browser.

No NameServer version of the request round-trip

The NameServer is used for load balancing, fault tolerance, and location independence of OpenEdge server applications. These are detailed in Chapter 5, "Configuration of the NameServer." The NameServer uses UDP as a network protocol and some network administrators do not want UDP on their networks for various reasons, some justified and others not. Progress introduced a "No NameServer" connection procedure for OpenEdge server applications in Version 9.1D. This will remove Step 0 through Step 2 of the connection process above.

To achieve this, configure the AppServer or WebSpeed broker so that it does not register with a NameServer by unchecking the **Register with NameServer** check box in the **Controlling NameServer**, as shown Figure 2–3. The **Registration Mode** is irrelevant for a No NameServer configuration. The example shows an AppServer configuration, but WebSpeed is identical.

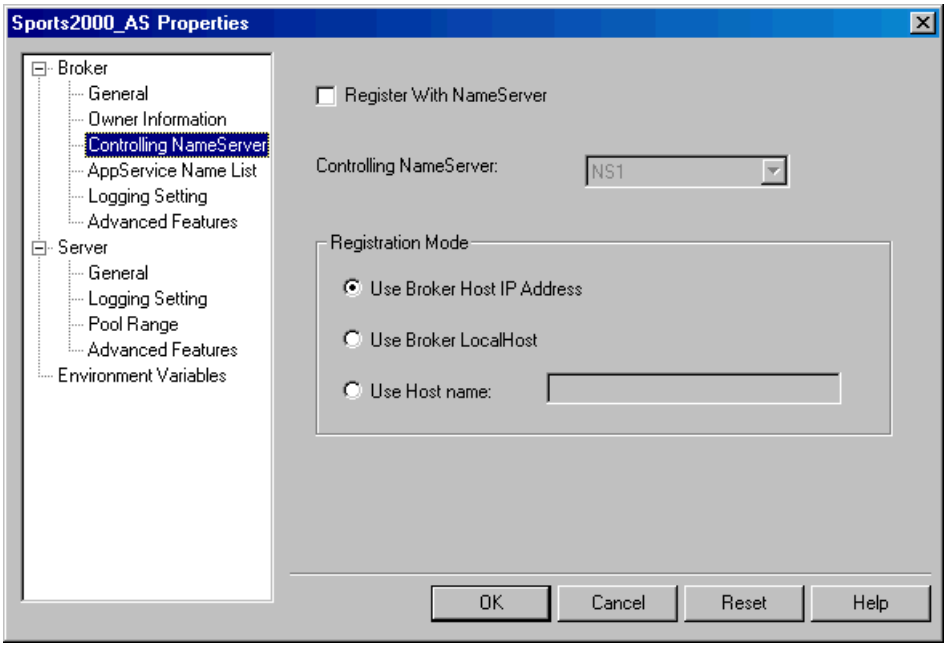


Figure 2–3: Configuring No NameServer

When using WebSpeed, the WebSpeed Messenger must point directly to a WebSpeed broker. To configure this, use the `cgiip.wsc` or `wspd_cgi.sh` mapping, as described in the "Hiding the CGIIP executable name from the end user " section on page 3–8. In this case, change the file contents so that you do not use the `-i wsbroker1` format for referencing the broker, as in the example, but rather the `myhost 3090` format, where `myhost` is the hostname or IP address for the WebSpeed broker machine and `3090` is the port number of the broker. It also means that you cannot use the `WService=...` on the URL.

If you are using the AppServer, the AppServer client application must use a different connection string. The following example shows the NameServer option. The NameServer host is `zeus` and the port the NameServer is listening on is `5162`. The requested service is `inventory`:

```
hAppSrv:CONNECT ("-AppService inventory -H zeus -S 5162").
```

The following example shows the No NameServer version. In this case, `apollo` is the AppServer Host and `8911` is the AppServer broker port.:

```
hAppSrv:CONNECT ("-DirectConnect -H apollo -S 8911").
```

Firewall configuration and debugging

If you are deploying a public WebSpeed application or an AppServer Internet Adapter-enabled AppServer application, then you should be using firewalls to minimize the risk of network intrusions, as detailed in the "Firewalls" section on page 3–21.

The following sections explain how to configure a firewall to allow WebSpeed to function and how to debug a nonworking firewall deployment.

Firewall configuration

Using a firewall opens up more configuration issues because you need to configure the firewall to allow communications between the OpenEdge server host machines on particular ports using TCP or UDP protocols. In the "Request round-trip process" section on page 2–6, the entire round-trip request is shown. All of these messages need to go through the firewall. Different firewall configurations are shown in the "Examples of simple and complex configurations" section on page 3–35.

Figure 2–4 illustrates which ports need to be open and what protocols the messages use. (If you are using multiple firewalls, as in Figure 3–12, then the same principles apply.)

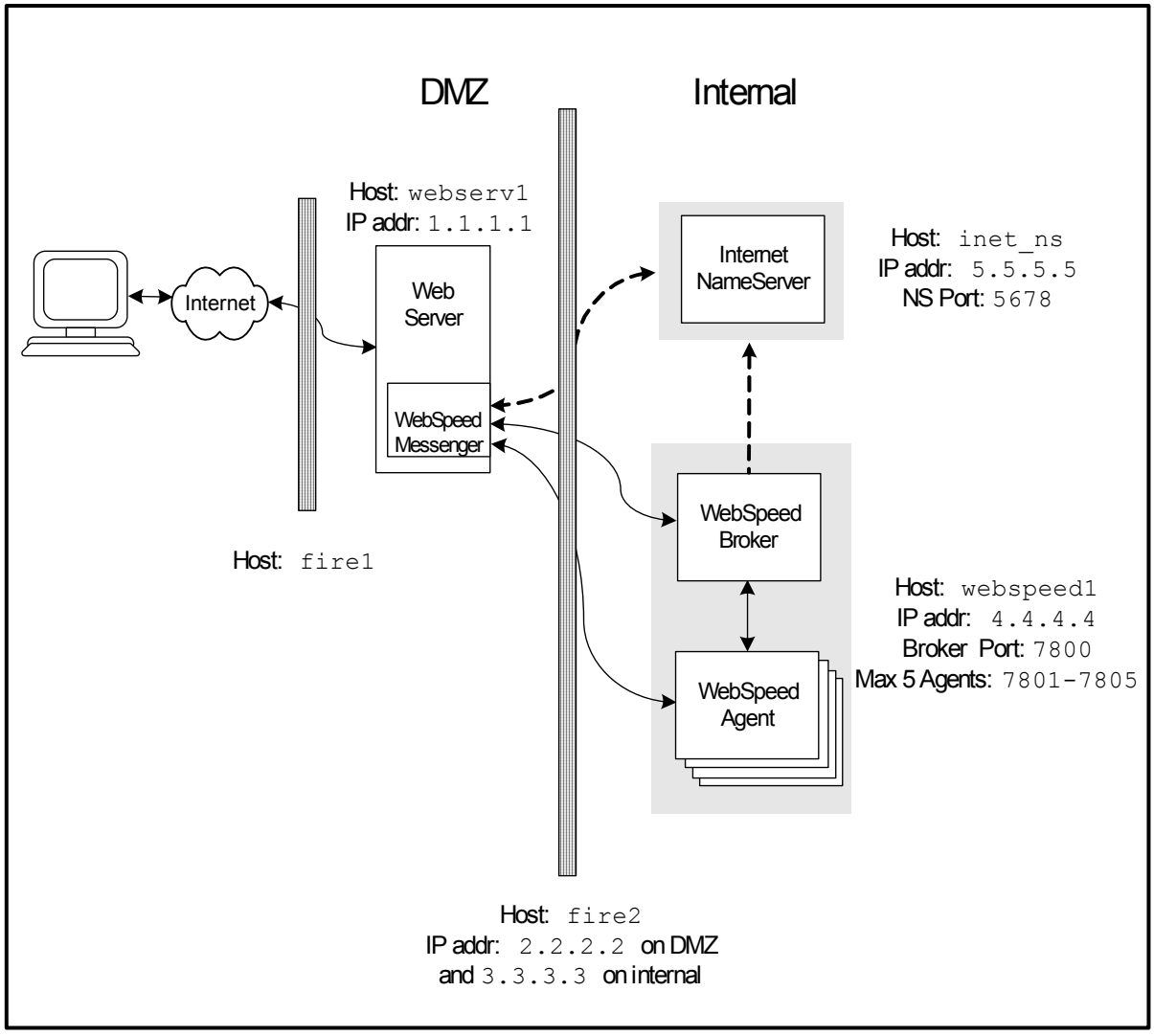


Figure 2–4: Firewall configuration

Note: The firewall `fire2` has two network cards, one for the Demilitarized Zone (DMZ) and one for the Internal network. Each of these has its own IP address, as shown.

In the following sections the hosts file is mentioned. In UNIX or Linux, this is located at /etc/hosts and in Windows NT and higher at C:\WINDOWS\system32\drivers\etc\hosts.

The WebSpeed Messenger ubroker.properties file must have the minNSClientPort and maxNSClientPort settings modified in the [WebSpeed.Messengers] configuration, as shown in Example 2–1. The port range must be big enough to cope with all the potential simultaneous requests from the Internet. In this case, there are 20 ports available. You can make this range bigger if needed. Also, you must change the setting for the NameServer to point to the correct host.

Example 2–1: Configuring ubroker.properties file for firewall

```
[WebSpeed.Messengers]
.
.
    minNSClientPort=5680
    maxNSClientPort=5699
    controllingNameServer=InternetNS
.
.

[NameServer.InternetNS]
.
.
    hostName=inet_ns
    location=remote
    portNumber=5678
.
.
```

You will need to configure the following:

- Between the Internet and Web server webserv1:
  - Allow inbound and outbound traffic from the Internet on port 80 (the default for HTTP) to the Web server. This is a standard configuration on most firewalls. If you are using HTTP/S (HTTP over SSL), then the default port is 443.
- Between the Web server (Messenger) and NameServer:
  - Allow UDP from IP Address 1.1.1.1 to 5.5.5.5 on port 5678. This is the inbound NameServer request traffic.
  - Allow UDP from IP Address 5.5.5.5 to 1.1.1.1 on ports 5680 to 5699 inclusive (assuming the above settings in the ubroker.properties file). This is the NameServer response traffic.

- Between the Web server (Messenger) and WebSpeed broker:
  - Allow TCP from IP Address 1.1.1.1 to 4.4.4.4 on port 7800 for the inbound request.
  - Allow TCP from IP Address 4.4.4.4 to 1.1.1.1 on port 7800 for the outbound reply.
- Between the Web server (Messenger) and WebSpeed agents:
  - Allow TCP from IP Address 1.1.1.1 to 4.4.4.4 on ports 7801 to 7805 inclusive for the inbound request.
  - Allow TCP from IP Address 4.4.4.4 to 1.1.1.1 on port 7801 to 7805 inclusive for the outbound reply.

Most firewalls will need to do this by using "port forwarding." This means that when the firewall receives a request from a host on a certain port in the DMZ, it is passed through to a particular host on the internal network. When the `webserv1` machine makes a request to the NameServer, it cannot see IP address 5.5.5.5 directly, and it has to pass the request to the firewall machine `fire2`. The firewall then makes the request on the internal network to IP address 5.5.5.5 on its behalf. When the response comes back from the NameServer to the firewall, the firewall will send it on to the Messenger on the DMZ network. As an analogy, think of the firewall as a language interpreter where the WebSpeed Messenger speaks English and the NameServer speaks German. The Messenger needs to talk to the NameServer but cannot do so directly, so it forwards the request to the interpreter who, in turn, makes a request to the NameServer on the Messenger's behalf. The response is given to the interpreter by the NameServer, who then forwards it to the Messenger.

This is achieved by setting the `hosts` file on `webserv1` to have the host `inet_ns` set to 2.2.2.2, as shown below. When the Messenger looks for host `inet_ns`, it uses the IP address 2.2.2.2, which is the firewall host `fire2`:

```
127.0.0.1       localhost
2.2.2.2         inet_ns
```

Note: You do not need to have an entry for `fire2` in the `hosts` file as the DMZ machines never communicate with it by name. They think they are communicating with other machines on the internal network.

Similarly, the Messenger cannot communicate directly to the WebSpeed server host `webspeed1` at IP address 4.4.4.4 either. So, another entry needs to be made in the `hosts` file to make the Messenger communicate with the firewall instead of the "real" host, as shown:

```
127.0.0.1       localhost
2.2.2.2         inet_ns webspeed1
```

Because of this, you cannot use the default setting for the WebSpeed broker's registration mode. The default is to use the broker host IP Address. If you do this, the NameServer will tell the Messenger to try to contact the broker on IP address 4.4.4.4, which is not a valid IP address in the DMZ, and it will appear as if the broker has not responded. You need to set the broker to register using a defined host name, in this case `webspeed1`. When the NameServer responds this time, it tells the Messenger to try to connect using the host name `webspeed1`. The Messenger asks the operating system on its host for the IP address of `webspeed1`. Since we set this to 2.2.2.2 in the `hosts` file, this is the address that is returned, and the Messenger will use it. The firewall will then get the request and pass it through. Figure 2–5 shows this configuration setting.

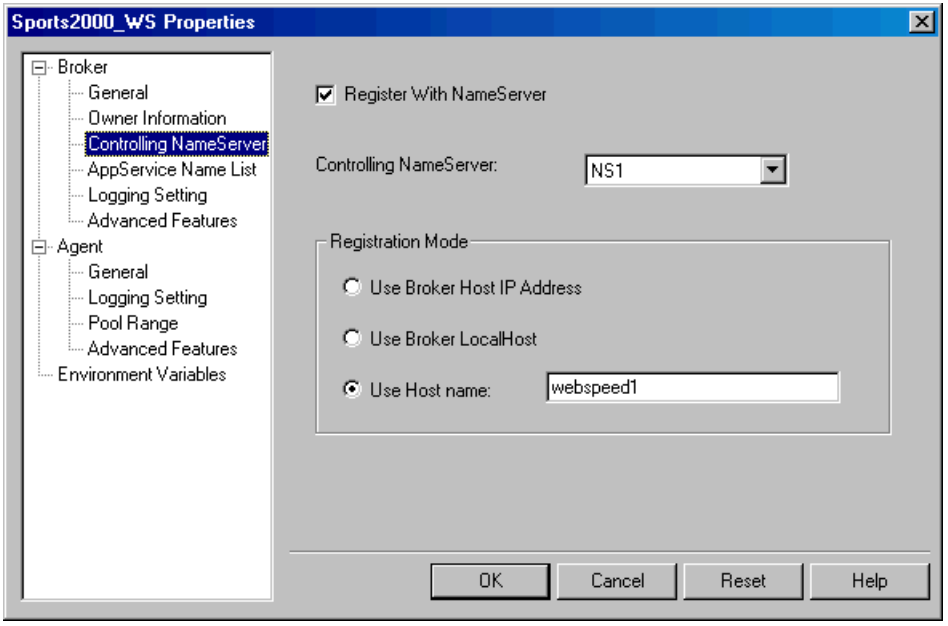


Figure 2–5: Setting host name

The NameServer and WebSpeed server hosts do not need the firewall IP address in their `hosts` file because they only respond to requests and do not make them.

Debugging firewall configurations

After configuring the firewall, you need to test the configuration to see if it works. The easiest way to do this is to try to run the WebSpeed application from the Internet. This probably means you need to disconnect the test client PC from the internal network and then dial an Internet Service Provider (ISP) to then act as a "real" internet client.

First, make sure everything works by entering the URL for the application into your Web browser. If it works, then you are lucky. Normally, the person configuring the firewall has left out one or two ports, or a `ubroker.properties` setting was left unchanged.

If you are like most people, the test failed. So, where is the problem? To trace where the issue lies, see the "Request round-trip process" section on page 2–6 to remind yourself what the entire roundtrip process is and test each stage one at a time. The error shown by the Messenger (if it worked that far) will lead you to the answer as well.

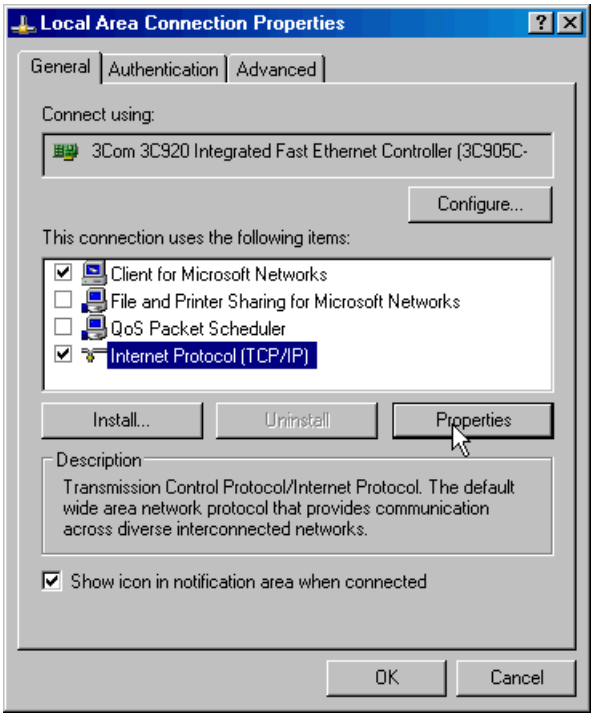
You might want to use a software tool like Ethereal (`http://www.ethereal.com`), to allow you to see what packets are traversing the network.

If you are using Microsoft Windows 2000 or later to host the Web server, you might find that UDP or TCP packets are being sent, but they are being ignored by the Web server machine. This can be caused by incorrectly setting the IP Packet filter. All ports used for the firewall access must be allowed in the IP Packet filter.

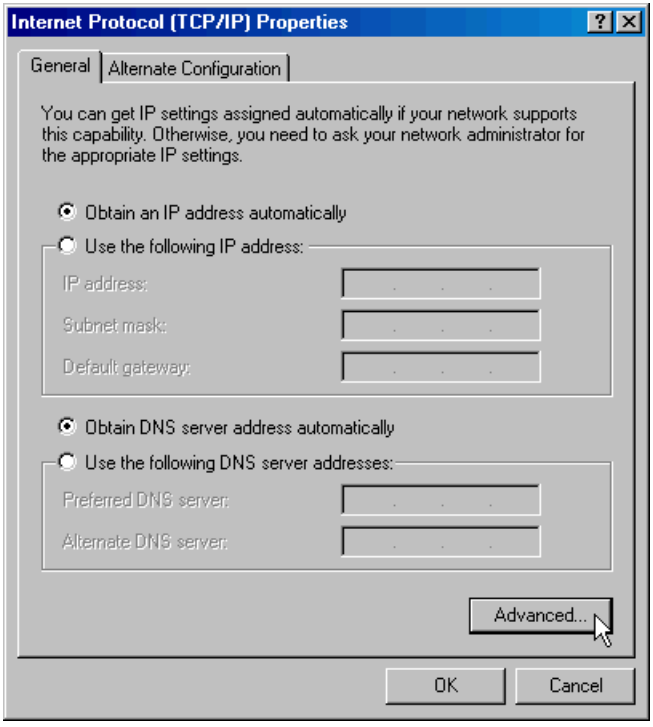
To access the IP Packet Filter settings:

1. In the **Windows Control Panel**, select the **Network Connections** icon.

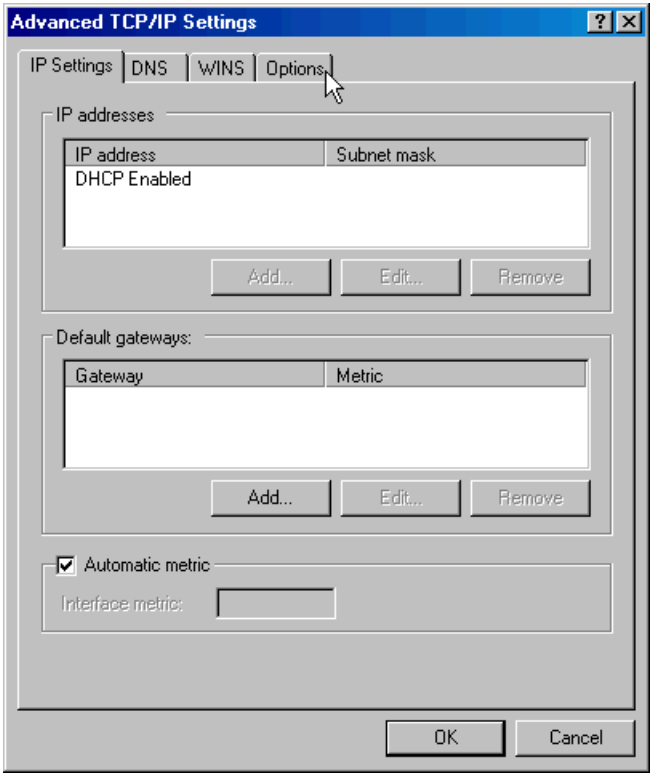
2. Right-click on your LAN connection and choose **Properties** from the pop-up menu. The **Local Area Connections Properties** dialog box appears:



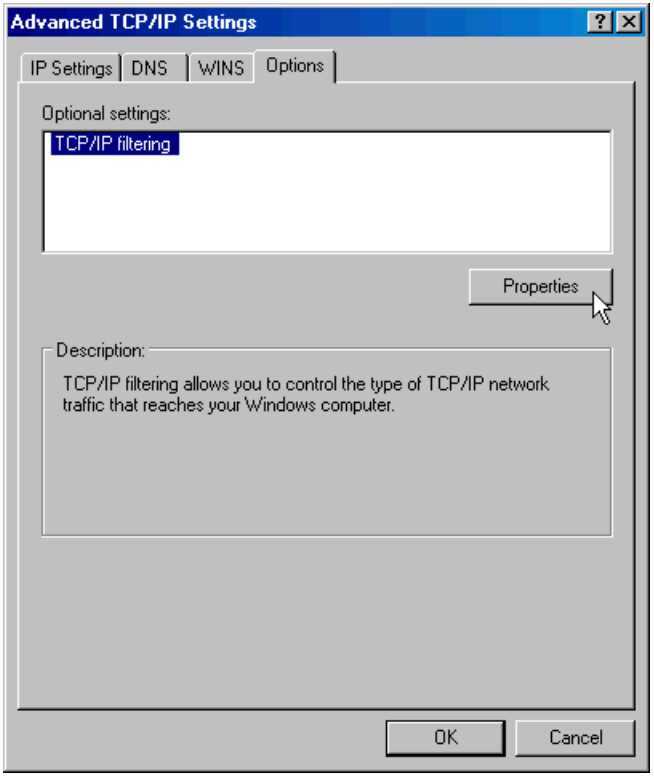
3. Select the **Properties** button. The **Internet Protocol (TCP/IP) Properties** dialog box appears:



4. Select the **Advanced** button. The **Advanced TCP/IP Settings** dialog box appears:

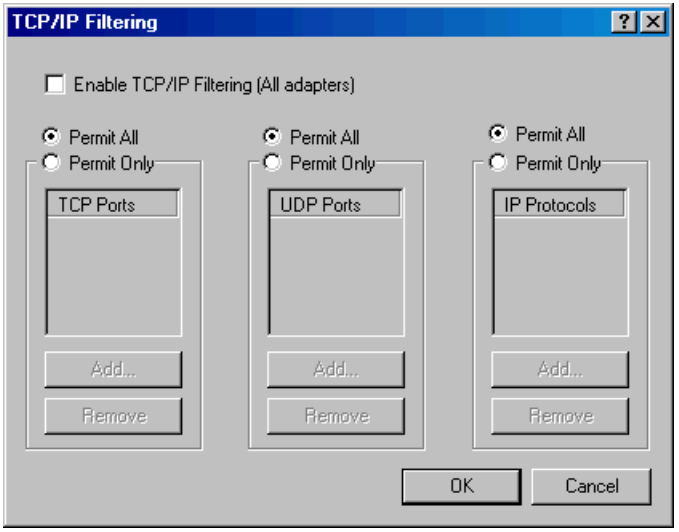


5. Select the **Options** tab:



6. Highlight **TCP/IP Filtering** in the list and then select the **Properties** button. The **TCP/IP Filtering** dialog box appears.

7. You can set the filter to allow all packets as shown, or you can restrict the ports allowed by adding them into the appropriate areas:



If you need to use DNS, then you also need to allow UDP port 53 and TCP port 53. For the Web server, you need port 80. For HTTP/S, you need port 443.

Web server access

Can your Web browser access the Web server? Put a test HTML file in the Web server's root directory to see if you can access it with `http://webserver/test.htm`. If you can, then delete the test HTML file and move on. If the file does not appear, check to see if the Web server is running.

WebSpeed Messenger

Does the WebSpeed Messenger run? If you get a "WebSpeed error from messenger process (6019)" error message, then the WebSpeed Messenger is running. If not, you should enable the Messenger logging function in `ubroker.properties` as shown in the excerpt below. The default logging level is 1, which is Errors Only. This setting should suffice to show the issues:

```
[WebSpeed.Messengers]
    ...
    logFile=@{WorkPath}\msgr.log
    loggingLevel=1
    ...
```

You may have received a Web server internal error. This is usually caused by a Web server misconfiguration related to the “executability” settings for CGI programs. Check your Web server documentation to make sure you have configured it to run your WebSpeed Messenger correctly.

If you use the Messenger Administration tool, you can test the configuration of your WebSpeed application. This is covered in the “Minimizing access to the WebSpeed Messenger Administration tool” section on page 3–10.

NameServer Access

If the Messenger is working, the next step is to confirm that the NameServer is being accessed. Set the NameServer’s logging level to 3 (Verbose) using the Progress Explorer or by editing `ubroker.properties` manually. To make this take effect, you will need to stop and restart the NameServer and wait for the WebSpeed broker to inform the NameServer that it is available. If the NameServer does not know about a service, it cannot direct clients to it. To check the NameServer to see if it knows about the WebSpeed server, use either the **Status** button in the Progress Explorer as shown in Figure 2–6 or the code `NSMAN -name NS1 -query` as shown in Example 2–2.

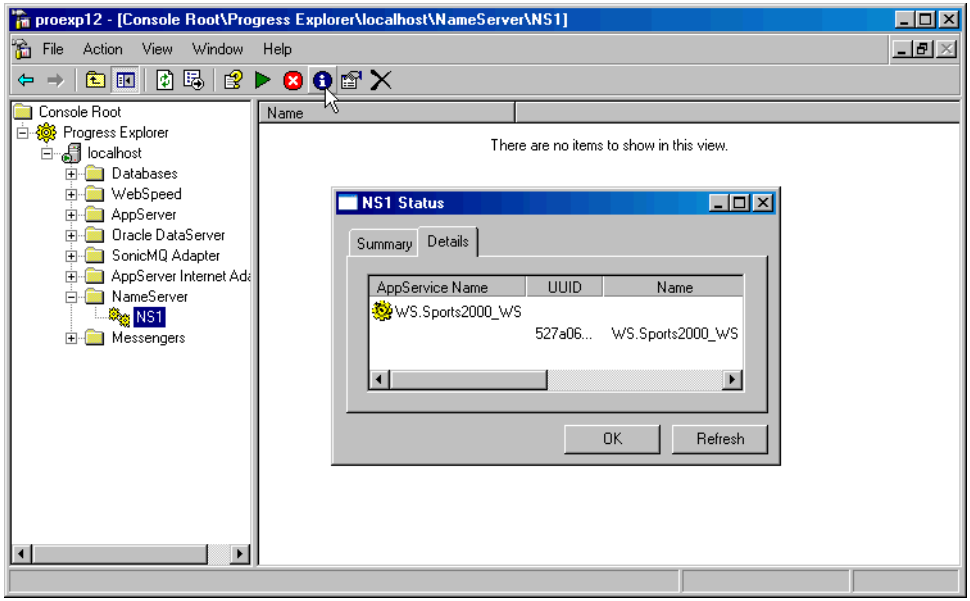


Figure 2–6: Checking NameServer access using the Progress Explorer

Example 2–2: Checking NameServer access using NSMAN -name NS1 -query

```
C:\>nsman -name NS1 -query
OpenEdge Release 10.0B as of Tues Apr 27 00:31:00 EDT 2004

Connecting to Progress AdminServer using rmi://localhost:20931/Chimera (8280)
Searching for NS1 (8288)
Connecting to NS1 (8276)

NameServer NS1 running on Host nexus Port 5162 Timeout 30 seconds.
Application Service             UUID            Name            Host
 Port   Weight  Timeout

WS.Sports2000_WS
        527a0623fe008210:67d940:f6484c1312:-7d87        WS.Sports2000_WS
nexus/192.168.123.121     3055   0       30
```

Now, try to access the application again. After the error is returned to the WebSpeed Messenger, check the NameServer's log file. You should see something similar to the following:

```
Thread-0>(26-Jul-03 18:42:15:107) Request received from 192.168.123.110 2167
for WS.Sports2000_WS. (8201)
Thread-0>(26-Jul-03 18:42:15:107) AppService = WS.Sports2000_WS Found = true
Number Of Brokers = 1. (8206)
Thread-0>(26-Jul-03 18:42:15:107) Response sent to 192.168.123.110
```

If you do not see the "Request received" in the log file, then the firewall is losing the inbound NameServer request. Otherwise, the outbound request is being lost. After debugging this stage, make sure to reset the logging setting.

Accessing the WebSpeed broker

This time, set the WebSpeed broker's logging setting to **verbose** and make the request. You should see something similar to the following at a time just after the NameServer log entry:

```
L-3055>(26-Jul-03 18:57:53:816) Received connection:: (8125)
C-0001>(26-Jul-03 18:57:53:836) Client connected : . (8533)
C-0001>ubWSclientThread.processConnRsp(): ubRsp = 0, getNeedNewConnID() =
false
C-0001>(26-Jul-03 18:57:53:836) The client C-0001 has disconnected from the
broker. (8084)
C-0001>(26-Jul-03 18:57:53:836) Client disconnected : . (8534)
```

If the WebSpeed Messenger error says it could not contact the broker, but the broker log file says it was contacted, then the fault is on the return path. If there is no contact logged in the broker log file, then it didn't receive the message. Either of these will point to the firewall rule that was left out or misconfigured.

Accessing the WebSpeed agent

Use the 4GL Trace function to see if the agent received the request. Configuring this feature is covered in the "Using other techniques to debug WebSpeed applications" section on page 3–32.

General notes on debugging

Think through the request process and see what the error messages say. This will lead to the issue most of the time.

Check the log files of the firewall itself. These will show what messages are flowing through it. You will probably have to filter these because a production firewall will have more than just your WebSpeed requests going through it.

Always use a new Web browser window for each test request. This is due to most browsers trying to speed up requests caching information. This can also be achieved by using the "Reload" function of the browser. See your browser document for information on setting your browser to not use cached copies of pages.

IP issues

The following sections cover two common IP issues:

- Domain Name System
- Multi-homed servers (multiple IP address servers)

Domain Name System

The Domain Name System (DNS) is one of the very useful, but often misconfigured, parts of any IP network. The job of the DNS is to resolve host names into IP addresses (and occasionally the reverse). If you misconfigure the DNS, then you will get some odd problems. The most common is that the client is taking a very long time to connect to the AppServer. This is probably due to a DNS issue where the DNS lookup is going outside your local LAN and onto your Internet Service Provider's DNS servers.

If you are experiencing any problems at connect time, it is very important to check that your DNS is configured correctly. To do this, go to a prompt (either UNIX or a Windows Command prompt) and enter `nslookup servername`. The time it takes to respond is the DNS lookup time. It should be nearly instantaneous. If you are using a `hosts` file entry to "fix" this problem, then you really should fix the DNS problem and remove the entry in the `hosts` file.

Multi-homed servers (multiple IP address servers)

If you have a server that has more than one network card in it, it is probably going to have more than one IP address. You could have problems connecting to the AdminServer and other OpenEdge servers in this instance. An example of a multi-homed server is shown in Figure 2–7.

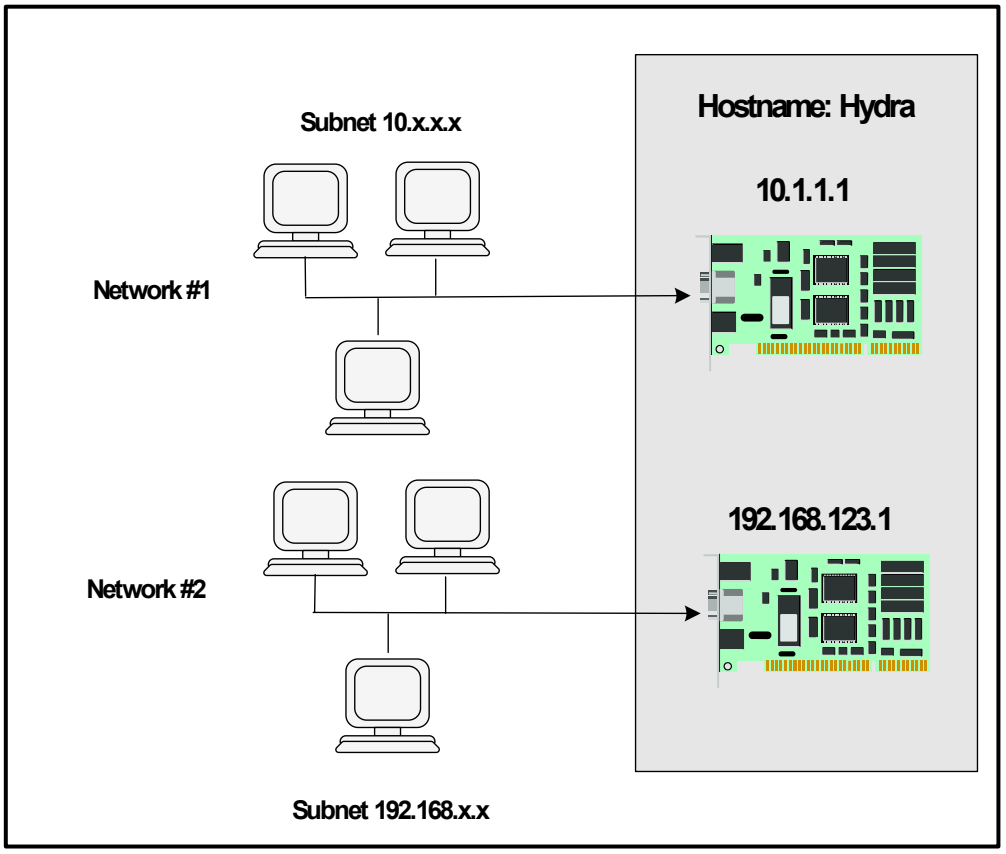


Figure 2–7: Multi-homed server

The clients on the 10.x.x.x subnet will not be able to access services on the host unless they use the correct IP address 10.1.1.1. Likewise, the 192.168.x.x subnet must use 192.168.123.1. To make all these clients connect to a single AppServer on the host, set up the `hosts` file on the machines in the 10.x.x.x subnet to read as follows:

```
127.0.0.1       localhost
10.1.1.1        hydra
```

And, on the 192.168.x.x subnet to read as follows:

```
127.0.0.1       localhost
192.168.123.1   hydra
```

All AppServer and WebSpeed instances running on `hydra` must have their **Register with NameServer** settings changed so that they do not register with an IP address, but instead with a host name, as shown in Figure 2–8.

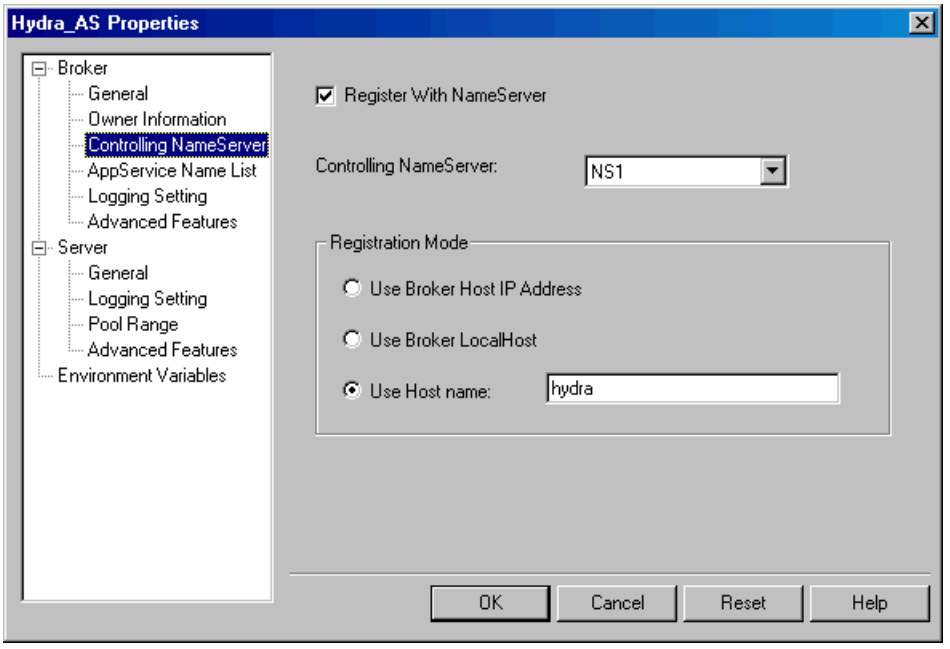


Figure 2–8: Register with NameServer setting

This enables the NameServer to tell the clients, regardless of what subnet they are on, to connect to the machine called `hydra`. It is then up to the client machine to decide what the appropriate IP address is.

3

Configuration of WebSpeed Servers

This chapter discusses how to configure WebSpeed servers. It also covers some general configuration tasks that also apply to the AppServer. The following topics are discussed:

- WebSpeed Messenger installation
- General WebSpeed server configuration
- WebSpeed security
- Error handling and debugging
- Using the AppServer to access the business logic
- Examples of simple and complex configurations

WebSpeed Messenger installation

You should always have a separate Web server and WebSpeed server when you are deploying an Internet-based WebSpeed application because of security reasons, as covered in the "WebSpeed security" section on page 3–4.

You need to install just the WebSpeed Messenger on the Web server. To do this, you will need to download the appropriate WebSpeed Messenger installation software, serial number, and control codes from the Progress Software Electronic Download Center. This is located at `http://www.progress.com/esd`. You need a valid Progress Software product serial number and control codes to download software from this site. You can also download Progress Software Service packs from this site.

Note: The version of the WebSpeed Messenger must exactly match the version of the WebSpeed server you want to use.

If your WebSpeed server is the same platform as your Web server, use the OpenEdge Installation CD to install the WebSpeed Messenger. You will need a valid serial number and control codes to install the WebSpeed Messenger from the CD.

After you have installed the Messenger, you must configure the `ubroker.properties` file in the *`OpenEdge-install`*`/properties` directory. If this file does not exist, there is a sample file called `msngrs.properties`. Copy this file to `ubroker.properties`.

General WebSpeed server configuration

When you are configuring a WebSpeed server, it is good practice to keep the port range for the agents to the smallest possible range. If you are configuring a WebSpeed server to have up to five agents, then the port range for the agents should be five.

Figure 3–1 shows an example of a WebSpeed server configuration with only five agents. Note the values of the **Minimum** and **Maximum port numbers**. If you set this port range too small or if another process uses a port in this range, the WebSpeed broker tries to launch a WebSpeed agent, the agent tries to use one of the currently in-use ports, and fails to start.

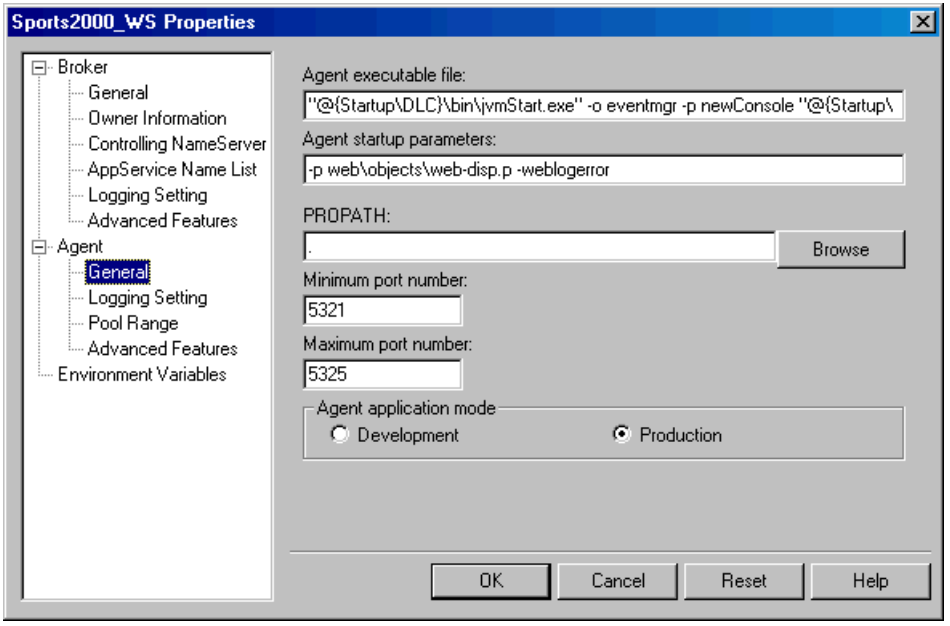


Figure 3–1: WebSpeed configuration with five agents

Configuring logging

There are three different logging settings for the WebSpeed broker: **Error Only**, **Terse**, and **Verbose**. The default is **Terse**, which provides enough information to be practical. You should leave the setting at **Terse**, unless you need to debug access to this WebSpeed server. Later in this manual, when we discuss how to configure a firewall-enabled deployment, **Verbose** logging is used.

The WebSpeed agent has four settings for logging: **Error Only**, **Terse**, **Verbose**, and a "hidden" level of 4GL Trace. 4GL Trace can only be set by using a text editor and modifying the `ubroker.properties` file, as discussed in the "Using other techniques to debug WebSpeed applications" section on page 3–32.

Managing log file size

The log files for the broker and agents can become quite large, especially during debugging sessions. You can view the log files using the Fathom Management console and use Fathom's File Monitoring feature to alert you if these files get too big.

To keep the log files to a manageable size, there are two options:

- Overwrite the log file every time the WebSpeed server is started. To do this, you deselect the **Append to … log file** check box on the **Broker and Agent Logging Setting** tab.
- Shut down the WebSpeed server, archive the current log file by moving it somewhere safe, and then restart the WebSpeed server. This is the recommended option because you can lose valuable historic information from the log file if it is overwritten each time the server starts. If you periodically shut down the WebSpeed server for maintenance, this is the opportune time to do the archiving. Furthermore, you can use the Fathom Job features to automate the process of archiving log files.

WebSpeed security

When using WebSpeed to allow access to your enterprise database over the Internet, the issues that should be foremost in your deployment plan are security, closely followed by performance. Providing access to your corporate data for "unknown" users without providing adequate security is akin to leaving the computer room door unlocked and displaying a sign saying "Please Steal Me" on the door.

Many people think that using a firewall alone will provide adequate security. However, using a firewall is just the first small step in providing secure access to your data.

To fully secure your deployment, you also need to secure your network traffic, your Web server, your WebSpeed server, and lastly, your application. Each of these topics is covered in detail in the following sections.

Securing your network traffic

When accessing a Web site, the content of the Web page that is returned is sent across the network in plain text. If you have a network sniffer, either a hardware device or software tools like Ethereal (http://www.ethereal.com), you can capture all the network traffic that passes your device.

If your Web site is on the Internet, a technician sitting in your Internet Service Provider's site can see the data passing between your Web server and whoever is accessing it. If your Web site is an Intranet, then anyone on the same physical network, in other words, most employees, can see the data.

If the data is private or confidential, then you should secure it from prying eyes. On an Internet site, confidential information might be credit card details or customer information; on an Intranet site, it might be salary details or bank account information if you are using a Web-based HR/Payroll application.

Securing this traffic is fairly easy; you need to enable HTTP/S or Secure Socket Layer (SSL) for HTTP on your Web server. HTTP/S encrypts the data flowing between your Web server and the client process (normally a Web browser) using Public Key Cryptography. You will need a Digital Certificate to allow this encryption to take place.

You can purchase SSL certificates for a public Web site from Verisign (http://www.verisign.com), Thawte (http://www.thawte.com), GeoTrust (http://www.geotrust.com), and others. These sites also have documentation explaining SSL and the process for purchasing, installing, and configuring SSL on many popular Web servers.

You should purchase the highest level of encryption possible for your locality. Most countries now allow 128-bit SSL, while some are still limited to 40-bit. The Digital Certificate provider will let you know the highest level that you can purchase.

If you are hosting a private Web site or an Intranet, then you can generate your own certificates. This has the benefit of being free, but the users of your site will have to accept their Web browser's warning that the certificate from your site is not trusted. To generate your own certificates, see your Web server's documentation.

After you have enabled SSL, you can use `https` instead of `http` as the URL protocol for your Web site, and then the data will be encrypted. For example, if your Web site address was:

```
http://www.mysite.com
```

You can now use:

```
https://www.mysite.com
```

Securing your Web server

Since your Web server is the first computer that users access, it is also the first machine you should start securing. From a security point of view, there is much discussion regarding which platform is better to run a Web server on, either Windows or UNIX/Linux. The reality is that both platforms have bugs that need to be patched. Do not get a false sense of security by using one platform over the other. Do not run an "out-of-the-box" version of either platform. Also, make sure that any operating system and application patches applicable to your Web server are applied as soon as they become available.

For a publicly accessible Web site, you should minimize the other services running on this machine. This provides better security, as the fewer things running on this machine, the fewer things can go wrong or be compromised.

You should also read all the Web server's documentation that deals with security. Most Web servers ship with most security settings disabled. You should go through all the settings and turn off any Web server features that you do not need.

Hiding your Web server type and version

It is good practice to hide the "brand" and version of your Web server process to make it harder for "script-kiddies" to find out which Web server you are using.

To see how your Web server responds, use a Telnet session to access the port that the Web server is listening to. The default port is 80. The following procedure shows the commands to type. Replace the ***hostname*** with your Web server's name. You might find that when you type `GET / HTTP/1.0` it might not be echoed back to you:

To check your Web server response:

1. Type **`telnet hostname 80`** and press **ENTER**.
2. Type **`GET / HTTP/1.0`** and press **ENTER** twice.

Note: Be sure to type a space preceding and following the first **`/`** in the **`GET / HTTP/1.0`** command.

The following is echoed back to you:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Fri, 11 Jul 2003 16:59:53 GMT
Content-Type: text/html

. . . HTML text of the default page . . .
```

In the previous example, you can easily see that the Web server is Microsoft's Internet Information Server (IIS) Version 5.1.

If you can modify the HTTP headers, make the `Server` setting return a generic name, like `WebServer`. Consult your Web server's documentation to see if it is possible and how to modify the HTTP headers.

Changing your script directory names

You should not use the standard script directory names. If you have an Apache server, do not use `cgi-bin`. If you are using Microsoft's IIS, do not use `Scripts`. See your Web server documentation for instructions on how to create a different script directory.

Most Web servers also ship with default home pages, as well as demonstration scripts. These generally should be disabled or deleted.

Hiding the CGIIP executable name from the end user

Hiding the WebSpeed Messenger name from the end user also provides a level of security. When you access a WebSpeed application, the URL used will look similar to the following if you are using Windows as the Web server:

```
http://www.mysite.com/scripts/cgiip.exe/WService=Orders/main.r
```

If you are using UNIX, then it will look similar to the following:

```
http://www.mysite.com/cgi-bin/wspd_cgi.sh/WService=Orders/main.r
```

Using the default names is bad security practice because it lets people know what application server you are using, in this case WebSpeed. For example, if you perform a Google search for `wspd_cgi.sh` or `cgiip.exe`, you will find many sites using WebSpeed. Some of these are not securely deployed.

Microsoft IIS

If you are using Microsoft IIS, then Progress Software has shipped an example file explaining how you can hide the Messenger's name. It is called `cgiip.wsc` and by default is located in the `C:\InetPub\Scripts` directory. It is recommended that you rename the file to something that is meaningful only to you, for example, `orders.inet`. The extension (`.inet`) must be an unused extension on your machine. You should also delete the `cgiip.exe` and `wsisa.dll` Messenger files in the `Scripts` directory.

If you open the `orders.inet` file using a text editor, you will see instructions on how to configure IIS to run this script when it is part of the URL.

Note: If you are using IIS 4.x or 5.x, you might find that the **Configuration** button mentioned in the instructions is disabled. To enable the **Configuration** button, click on the **Create** button just above it.

Use the extension you have chosen (for example, `.inet`) instead of the `.wsc` extension mentioned in the instructions.

At the end of the newly created `orders.inet` file, change the WebSpeed service name from `wsbroker1`. For the example above, use `Orders`.

All lines beginning with `#` are comments. The only required line is the one that references the service name or host and port of the WebSpeed broker.

Assuming that you have changed the `Scripts` directory to be `web`, the URL would become:

```
http://www.mysite.com/web/orders.inet/main.r
```

If you have more than one WebSpeed service, then you will need a `.inet` file for each service.

UNIX

There are many different Web servers available for the UNIX platform. To find out which Web servers Progress Software has tested and certified, search the Knowledge Center. You can access the Progress Knowledge Center at the following URL: `http://www.progress.com/support/kb`.

Each of these has different configuration instructions. You should read the documentation supplied by the Web server vendor to determine how to enable CGI applications. Rename the Progress-supplied `wspd_cgi.sh` to something that is meaningful only to you and change the WebSpeed service name from `wsbroker1`. If you have changed the `cgi-bin` directory to `web` and allowed `.inet` scripts to be run as CGI programs, then the URL you would use is:

```
http://www.mysite.com/web/orders.inet/main.r
```

Minimizing access to the WebSpeed Messenger Administration tool

If the Messenger Administration tool is enabled, users can see your configuration information. This information can then be used to compromise your application.

To disable this feature, you can do one of two things: either disable the feature totally or allow only "trusted" IP addresses to access the Messenger Administration tool.

To totally disable the feature, edit the `ubroker.properties` file on the Web server and make sure that the `AllowMsngrCmds` is set to `0` (zero) in the `[WebSpeed.Messengers]` section and that it is not overridden in any of the `[WebSpeed.Messengers.CGIIP]`, `[WebSpeed.Messengers.WSASP]`, `[WebSpeed.Messengers.WSISA]`, or `[WebSpeed.Messengers.WSNSA]` sections.

To allow a list of IP addresses to access the Messenger Administration tool, edit the `ubroker.properties` file on the Web server and set the `AllowMsngrCmds` to `1` (one) and the `wsmAdmIPList` to a comma-separated list of IP addresses that are permitted to access the Messenger Administration tool. This needs to be done in the appropriate Messenger section: `[WebSpeed.Messengers.CGIIP]`, `[WebSpeed.Messengers.WSASP]`, `[WebSpeed.Messengers.WSISA]`, or `[WebSpeed.Messengers.WSNSA]`.

If the Messenger Administration tool is enabled, you can change the default WebSpeed Messenger Error Messages, as described in the "Error handling" section on page 3–24.

You can also verify your WebSpeed configuration. Use the following URLs to see the Messenger Administration tool:

```
http://www.mysite.com/scripts/cgiip.exe?WSMAdmin
```

or

```
http://www.mysite.com/cgi-bin/wspd_cgi.sh?WSMAdmin
```

Securing your WebSpeed server machine

Having secured your Web server machine, it is time to move one machine closer to the database. You now need to secure your WebSpeed server. This machine has, at least, an AdminServer and a WebSpeed server running on it. When you installed OpenEdge on this machine, you should have also enabled the AdminServer security mentioned in the "General OpenEdge server configuration tasks" section on page 2–2. You should make sure that all the vendor's security patches for this operating system have been applied, and check to see that the latest Progress Service Pack has also been installed. As you did with the Web server, you should also minimize other services running on this machine. This provides better security, as the fewer things running on this machine, the fewer things can go wrong.

The WebSpeed broker's configuration should also specify an owner. This allows the WebSpeed broker and agents to be started with the specified user's rights, not the root or system administrator's rights. See the "Broker ownership" section on page 2–4 for details.

You should always have a separate WebSpeed server for development/testing and production. These should also use different Web server machines and be assigned to different NameServers. If you do this, then the chance of outside access to the development machine is greatly reduced.

Figure 3–2 shows a deployment model that uses separate machines for the Internet production, intranet production, and the development/test servers. The databases are all installed on the same machine as the WebSpeed servers. This is the preferred route if your machine has the capacity to host both, as it will provide the best performance.

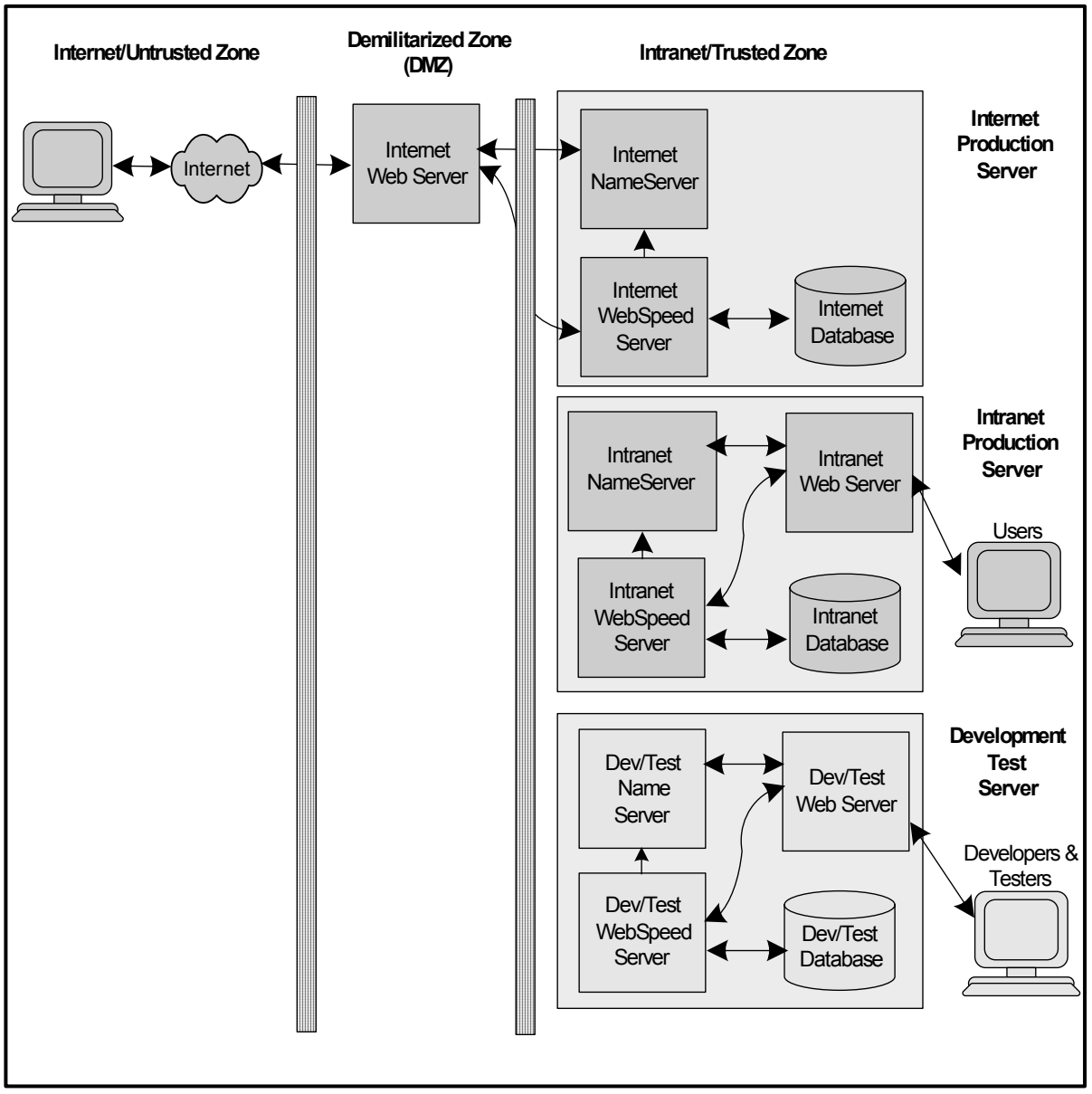


Figure 3–2: Deployment model with separate machines for the Internet Production, Intranet Production, and the Development/Test servers

Figure 3–3 shows a deployment model that uses two NameServers and puts all the production databases on one machine. This is useful because the Intranet and Internet applications might be using the same databases, and it lets you split the number of agents between Internet and Intranet access, saving license fees.

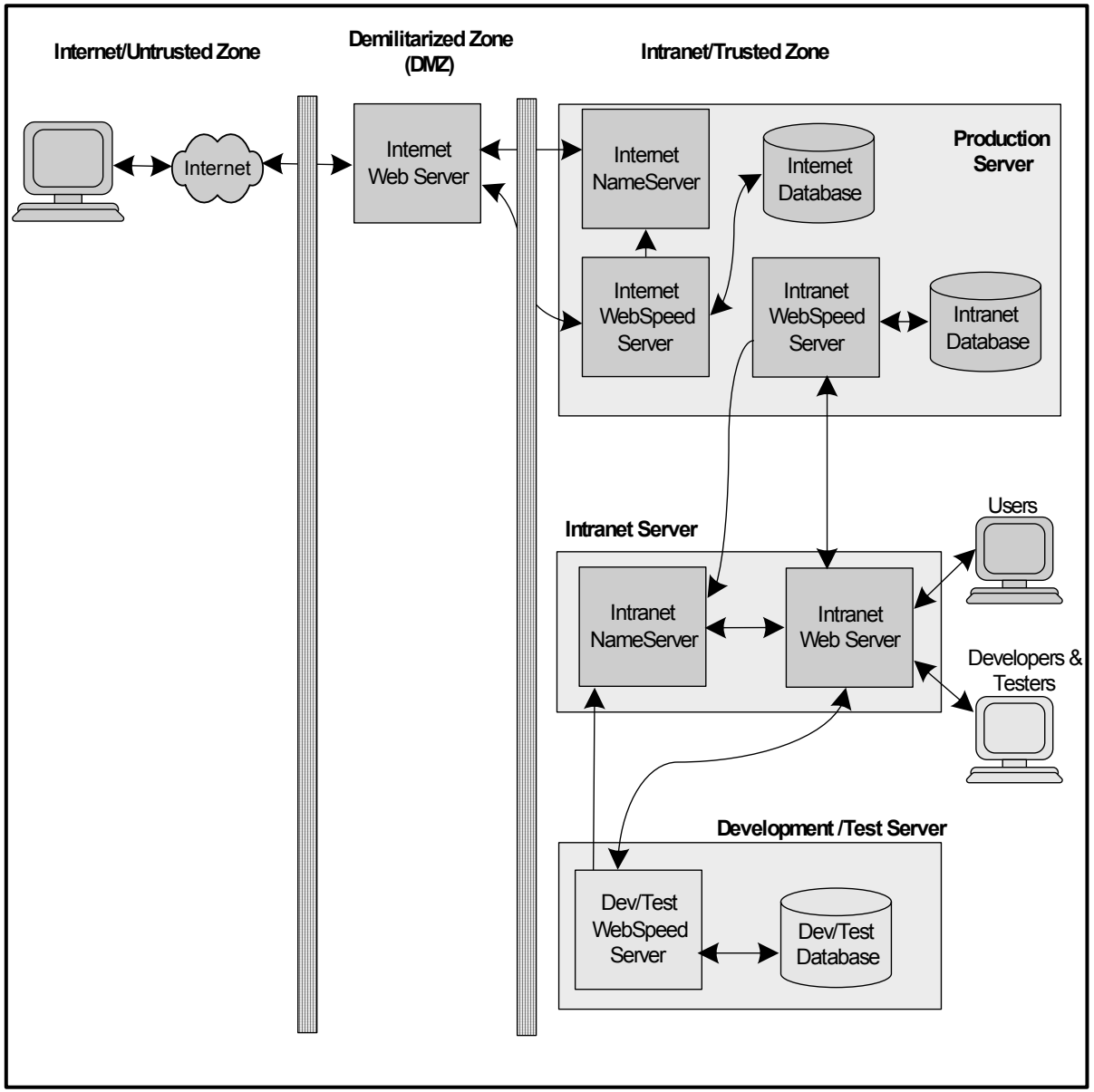


Figure 3–3: Deployment with two NameServers

All access from the Internet goes through the Internet NameServer, and all Intranet access (both production and development/test) goes through the Intranet NameServer. This means that the Internet NameServer only knows about the Internet applications and cannot hand requests to the Intranet production or development/test WebSpeed servers.

Using an AppServer to run your business logic allows you to place another level of indirection between your application and the database. This enhances the security of the application, as the WebSpeed server does not directly connect to the database; it accesses the data through the AppServer. See the "Using the AppServer to access the business logic" section on page 3–34 for information on how this can be achieved.

Securing your WebSpeed application

The application itself is probably the easiest place to make sure that it is secure. You should follow these rules to make sure that your application is as secure as possible.

Using DBAUTHKEY to lock your r-code to the database

An under-used feature of OpenEdge is the DBAUTHKEY (and RCODEKEY) features of PROUTIL.

With DBAUTHKEY, you assign a key to the database and then any code compiled against that database will have the key in it. When it comes time to run the code, if the key in the database does not match the key in the r-code, you will get an error similar to the following:

```
** CRC for table does not match CRC in program. Try recompiling. (1896)
```

If you already have r-code deployed, use the RCODEKEY function of PROUTIL to tag the existing r-code without the need to recompile.

See *OpenEdge Data Management: Database Administration* for more information on using the DBAUTHKEY and RCODEKEY features of PROUTIL.

Use the agent's production setting

For production environments, either Internet or Intranet, you should set the agent application mode to **Production**. You set this option in the **Properties** dialog box, as shown in Figure 3–4. After setting this, you should stop and start the WebSpeed service to activate the change.

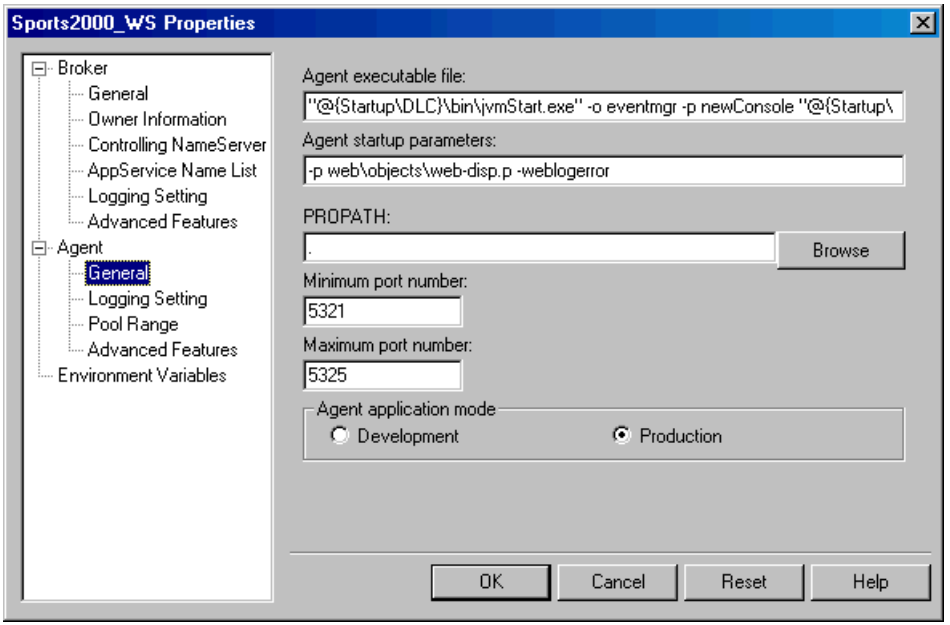


Figure 3–4: Setting Production mode for WebSpeed agents

Modifying web-disp.p

By default, the agents run `web/objects/web-disp.p` as their startup program. Each request that is issued to an agent runs through this code. This is the best place to control what happens to each request.

Modify `web-disp.p` to:

- Make sure that certain r-code can only be run by certain users.
- Turn off the `PING` or `DEBUG` facilities.
- Connect to a database every time a request comes through.
- Check for a user timing out.

Because each request must go through this code, any changes made to `web-disp.p` are system wide.

If you want to change this code, you should move it into your application's source tree and rename it. This way, when Progress Software ships a newer version of `web-disp.p` in a service pack, your changes will not be overwritten. You should also compare your code with the new code shipped in the service pack to make sure you also incorporate any bug fixes or enhancements.

Example 3–1 shows a simplified version of the **default** WebSpeed 3.1D (Progress 9.1D) `web-disp.p`.

Example 3–1: Default web-disp.p

```
/* Set the web-request trigger. */
ON "WEB-NOTIFY":U ANYWHERE DO:
  OUTPUT {&WEBSTREAM} TO "WEB":U.

  /* Parse the request/CGI from the web server. */
  RUN init-cgi     IN web-utilities-hdl.

  /* Initialize for web-request. */
  RUN init-request IN web-utilities-hdl.

  AppProgram = (IF AppProgram = "debug":U THEN "webutil/debug.p":U ELSE
                (IF AppProgram = "ping":U  THEN "webutil/ping.p":U  ELSE
                (IF AppProgram = "reset":U THEN "webutil/reset.p":U ELSE
                 AppProgram))).

  RUN run-web-object IN web-utilities-hdl (AppProgram) NO-ERROR.

  /* Run clean up and maintenance code */
  RUN end-request IN web-utilities-hdl NO-ERROR.

  /* Output any pending messages queued up by queue-message() */
  IF available-messages(?) THEN
    output-messages("all", ?, "Messages:").

  OUTPUT {&WEBSTREAM} CLOSE.
END. /* ON "WEB-NOTIFY" */


/* Wait for a web-request to come in */
WAIT-FOR-BLOCK:
REPEAT ON ERROR UNDO WAIT-FOR-BLOCK, LEAVE WAIT-FOR-BLOCK
       ON QUIT  UNDO WAIT-FOR-BLOCK, LEAVE WAIT-FOR-BLOCK
       ON STOP  UNDO WAIT-FOR-BLOCK, NEXT  WAIT-FOR-BLOCK:

  WAIT-FOR "WEB-NOTIFY":U OF DEFAULT-WINDOW.
END. /* WAIT-FOR-BLOCK: REPEAT... */
```

Note: Example 3–1 code will not run. Much of the code has been removed. The purpose of this example is to show program flow.

Example 3–2 shows a simplified, secure `web-disp.p`. Again, this code will not run; you would need to insert the **bold** text into the original `web-disp.p` replacing the "`AppProgram = ...`" code.

This code stops `PING`, `DEBUG`, and `RESET`, changes the extension of any requested program into r-code, checks that the r-code file exists, and verifies if this r-code is valid for this user by

looking up a database table called `UserPrograms`. You will need to create a table called `UserPrograms` containing (at least) both these fields. Also `UserID` is a variable that you must instantiate.

You would usually use a cookie, hidden fields, or URL parameters to hold the user's ID. This should be encrypted in a suitable manner. See the "Parameter passing" section on page 3–20 for an example of encrypting this ID.

Example 3–2: Secure web-disp.p *(1 of 2)*

```
/* Set the web-request trigger. */
ON "WEB-NOTIFY":U ANYWHERE DO:
  DEFINE VARIABLE vLocn AS INTEGER NO-UNDO.

  OUTPUT {&WEBSTREAM} TO "WEB":U.

  /* Parse the request/CGI from the web server. */
  RUN init-cgi     IN web-utilities-hdl.

  /* Initialize for web-request. */
  RUN init-request IN web-utilities-hdl.

  /* Remove current extension */
  vLocn = R-INDEX (AppProgram, ".").
  IF vLocn > 0
  THEN
    AppProgram = SUBSTR (AppProgram, 1, vLocn - 1).

  /* Add a .R */
  AppProgram = AppProgram + ".r".

  /* Can this User run this program OR does it exist? */
  IF NOT CAN-FIND (UserPrograms WHERE UserPrograms.UserID  = UserID
                                  AND UserPrograms.Program = AppProgram)
     OR
     SEARCH (AppProgram) = ?
  THEN
    AppProgram = "NotValidProgram.r".
```

Example 3–2: Secure web-disp.p *(2 of 2)*

```
 RUN run-web-object IN web-utilities-hdl (AppProgram) NO-ERROR.

  /* Run clean up and maintenance code */
  RUN end-request IN web-utilities-hdl NO-ERROR.

  /* Output any pending messages queued up by queue-message() */
  IF available-messages(?) THEN
    output-messages("all", ?, "Messages:").

  OUTPUT {&WEBSTREAM} CLOSE.
END. /* ON "WEB-NOTIFY" */
```

After you have created your `new-web-disp.p`, you need to change the agent parameters to reference it, as shown in Figure 3–5.

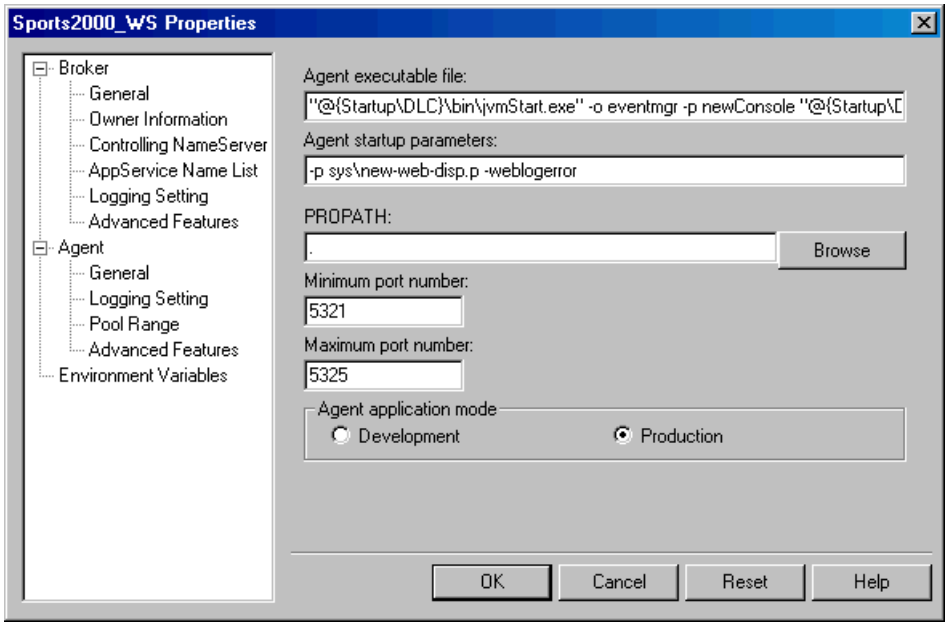


Figure 3–5: Changing agent parameters to reference web-disp.p

Minimize the PROPATH

It is essential that the PROPATH is kept to a minimum, both for performance and security. The *OpenEdge-install*/tty directory and all the r-code libraries (*.PL) in the *OpenEdge-install*/tty directory are added to the end of your PROPATH setting by default. This means that there are many programs in your PROPATH that you didn't write and anyone can run these programs by adding them to the end of your URL.

To avoid this, simply rename the *OpenEdge-install*/tty directory to *OpenEdge-install*/tty_save. Then, copy all the r-code files you use to a new directory called tty in your deployment area and add this to the end of your PROPATH. Remember that some of the r-code files WebSpeed might use are in the .PL files, and you will need to extract them using the PROLIB utility documented in *OpenEdge Deployment: Managing 4GL Applications*.

Parameter passing

If you want to pass parameters between Web requests, you can use hidden fields on forms, URL parameters, cookies, or a combination of each technique. Each technique has pros and cons. Hidden fields only work on forms, URL parameters are visible to the end user, and cookies are not allowed by some users.

The simplest way to pass many parameters between Web requests is to use the database. You pass a unique identifier for each user or session between requests, and use this as a key into a "state" table held in the database. This technique requires that only a small token be passed between requests, as the majority of the data is safe and secure in the database.

Do **not** pass the unique identifier in plain text. Doing so makes it very easy for an end user to change the value (even in hidden fields or cookies) and become someone else. Use code, similar to the code shown in Example 3–3, to prevent people from changing the unique identifier, unless they know the hidden words, in this case "Web" and "Speed."

Example 3–3: Passing unique identifiers

```
/*
** This code assumes that the Unique ID will not contain
** any colons (:).
*/

DEFINE VARIABLE vToken    AS CHAR NO-UNDO.
DEFINE VARIABLE vUniqueID AS CHAR NO-UNDO.

/* WebEncode function */
FUNCTION WebEncode RETURNS CHAR (pUniqueID AS CHAR):
  RETURN pUniqueID + ":" + ENCODE ("Web" + pUniqueID + "Speed").
END.
```

```
---- Use this to encode the Unique ID, then pass as parameter ----

/* Encode Unique ID */
vToken = WebEncode (vUniqueID).

---- Use this to decode the token passed as a parameter ----

/* Decode and check Token */
vUniqueID = ENTRY (1, vToken, ":").
IF vToken = WebEncode (vUniqueID)
THEN
  /* vToken has not been modified */
ELSE
  /* ERROR - vToken has been modified */
```

Firewalls

A firewall is the first line of defense for basic network security. It is usually a separate device that sits between the untrusted network (the Internet) and the trusted network (the Intranet). The role of a firewall is to stop unauthorized access of information in the trusted network by individuals on the untrusted network, but allow defined access from the trusted to the untrusted. An analogy for a firewall is a moat around a castle with the drawbridge being the firewall device. The drawbridge is controlled by guards who only allow certain traffic in, usually after inspecting it, and will allow outbound traffic if it has permission.

There is usually a third network called the DMZ or Demilitarized Zone. This network is separate from both the others, but it can communicate with both. This is a semi-trusted area that is protected by the firewall, so only certain traffic can come in. Any traffic coming from the DMZ into the trusted network has strict rules placed upon it, so errant requests are denied. There are three physical network ports on a DMZ-enabled firewall, one for each of the networks.

Figure 3–6 shows a firewall with a DMZ. This is the usual configuration for a firewall.

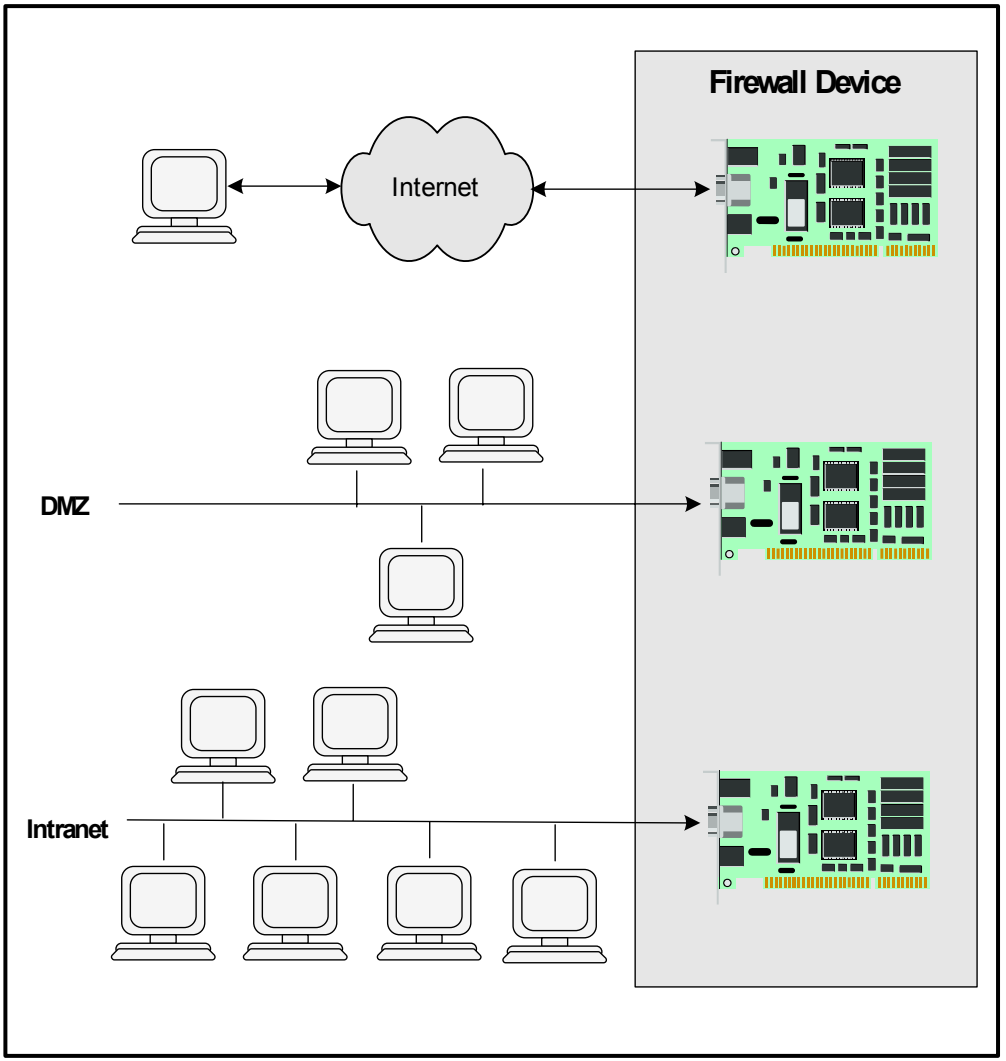


Figure 3–6: Firewall with DMZ

Figure 3–7 shows a more secure firewall configuration. The reason for having two firewall devices from different manufacturers is two-fold. First, having only one device means that any bugs or security holes in the firewall software could allow direct connection between the untrusted and trusted networks. Second, using different manufacturers' hardware/software combinations stops hackers from using the same exploit or security hole on both devices.

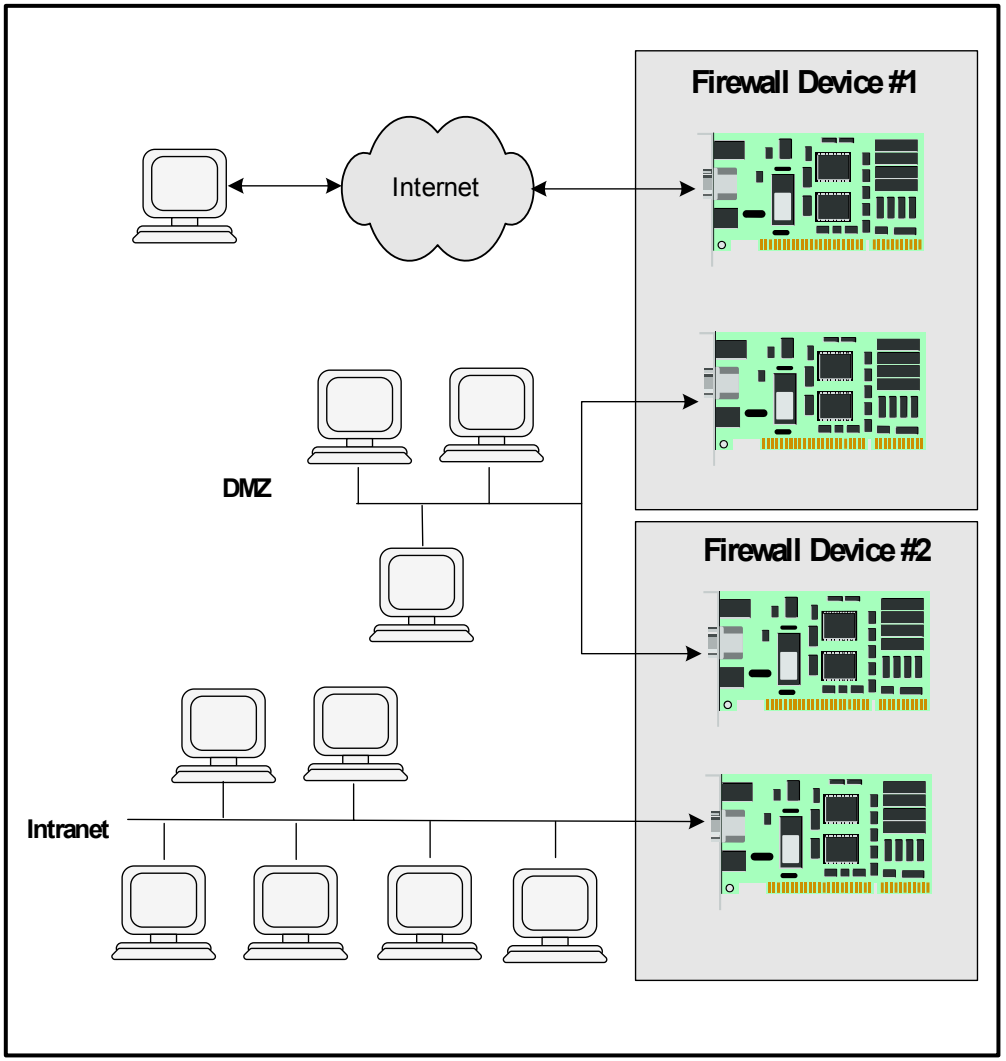


Figure 3–7: Secure firewall configuration

Firewalls can be implemented in either hardware or software. A hardware firewall is a machine that has a proprietary operating system and software for providing the service. Any patches provided by the firewall supplier should be applied as soon as possible to minimize the risk of attack.

A software firewall is a program that is loaded onto a general purpose computer, usually a PC, to provide the service. To be effective, software firewalls rely on the underlying operating system to be secure, so you should make sure that all the operating system manufacturer's patches are applied along with any updates to the firewall software. You should avoid running anything else on a software firewall's host machine. Some software firewalls do not use an underlying general purpose operating system; they use standard hardware, but load their own proprietary operating system along with the firewall software.

Error handling and debugging

When you are running a WebSpeed application in production, you should make sure that at no time does the user see any OpenEdge error messages, either from the WebSpeed Messenger or from your code. You must code your application in a manner that is very robust and will cope with varying input parameters. It is very easy to write code with WebSpeed, but it is a bit harder to write reliable, robust code.

Error handling

To make the WebSpeed Messenger return more "attractive" error messages, use the `WSMAdmin` option on the URL to configure the Messenger to redirect the user to a static HTML page for certain errors. At a minimum you should redirect the errors shown in Table 3–1.

Table 3–1: Redirecting WebSpeed error messages *(1 of 2)*

Message	Meaning
`Connection failure for host` *`hostname`* `port` *`portnumber`* `transport UDP. (9407)`	The messenger could not connect to the specified NameServer.
`NameServer at Host` *`hostname`* `Port` *`portnumber`* `is not responsive. (8239)`	The messenger could not connect to the specified NameServer.

Table 3–1: Redirecting WebSpeed error messages *(2 of 2)*

Message	Meaning
`Msngr: the specified service name does not exist or has a bad format. (5825)`	The NameServer could not find the specified service.
`Msngr: Disconnecting - all agents are currently busy, please try again later. (5832)`	The Messenger could not be allocated an agent to run the request.
`Msngr: Disconnecting with no header on WTA output web stream. (5814)`	The agent has died somehow. Usually caused by shutting the database down and not stopping the WebSpeed server.
`WebSpeed Agent Error: Agent did not return an HTML page. (6383)`	Your application failed somewhere before it even started to output the HTML. This can be due to records not being available before accessing them, a run-time error caused by a parameter mismatch, or a run-time error caused by a failing INT conversion.

To access the Error Customization Utility, use a URL similar to one of the following:

```
http://www.mysite.com/scripts/cgiip.exe?WSMAdmin
```

or:

```
http://www.mysite.com/cgi-bin/wspd_cgi.sh?WSMAdmin
```

The Error Customization Utility is documented in *OpenEdge Application Server: Developing WebSpeed Applications*.

Writing robust code

In your application you should always verify that the parameters passed are sensible. This is because the parameters passed on the URL, using hidden fields or cookies, can be modified by the user. For example, if you are expecting an integer parameter such as a page number, then the parameter might be passed on the URL like this:

```
http://www.mysite.com/web/orders.inet/show_results.r?page=3
```

If the user modifies the URL to have `xyz` instead of `3` as the page number parameter, your application might crash. To avoid this, you can use a section of code similar to the code in Example 3–4. You might also want to verify that the page number is within limits using the `MIN` and `MAX` functions.

Example 3–4: Robust code for passing parameters

```
DEFINE VARIABLE vPage AS INTEGER NO-UNDO.

ASSIGN vPage = INT (get-field ("page")) NO-ERROR.

IF ERROR-STATUS:ERROR /* Did the INT fail? */
THEN
  /* Sensible Default */
  vPage = 1.
ELSE
  /* Limit vPage to between 1 and vMaxPageNum */
  ASSIGN vPage = MIN (vPage, vMaxPageNum)
         vPage = MAX (1, vPage).
```

When writing WebSpeed applications, you should also make sure you check that records are available before accessing them, just as you do when using GUI, character, or batch programming.

Debugging the application

There are two main ways to debug an application: the first involves accessing the source code for the application and the second does not.

Using the OpenEdge Application Debugger to debug WebSpeed applications

To use the OpenEdge™ Application Debugger with a Windows version of WebSpeed, you need to be using the machine where WebSpeed is installed.

If you add the following lines into a WebSpeed application, the Debugger window will appear when these lines are executed:

```
DEBUGGER:INITIATE ().
DEBUGGER:SET-BREAK ().
```

Figure 3–8 shows the Debugger working in a CGI Wrapper program.

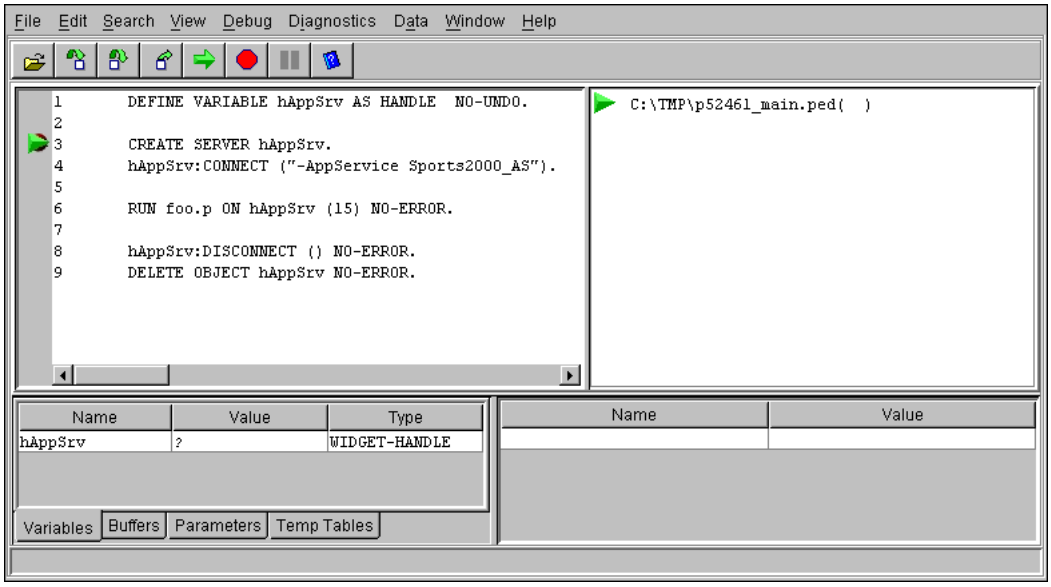
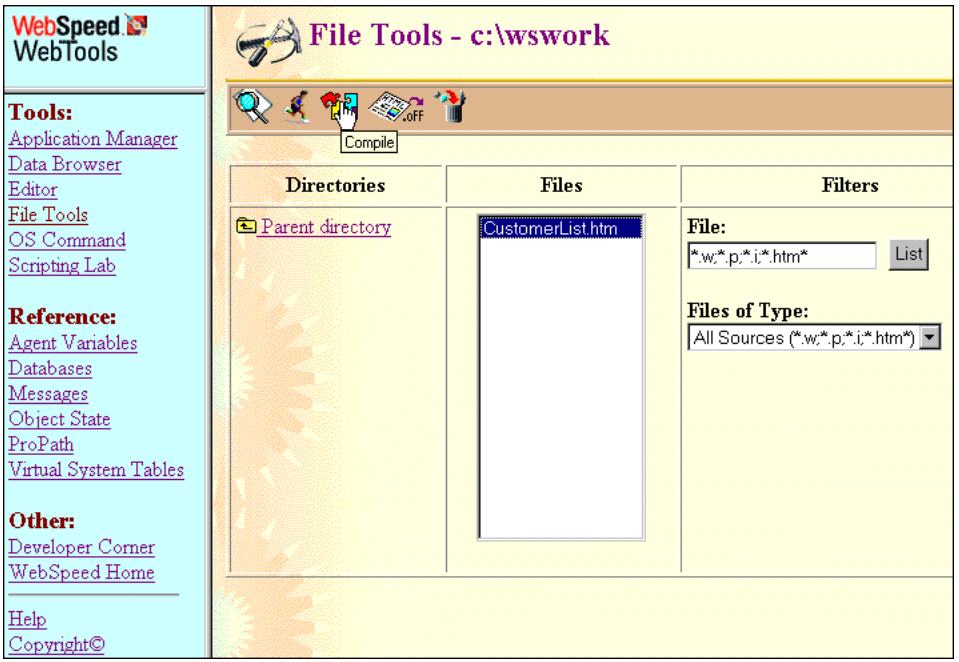


Figure 3–8: OpenEdge Application Debugger working in a CGI wrapper program

The Debugger will work with any WebSpeed development environment. If you use Embedded SpeedScript and you want to see source code in the Debugger, you need to compile the application in WebTools, as described below.

To see Embedded SpeedScript source code in the Debugger:

1. In WebSpeed WebTools, compile the application using the **Compile** option in **File Tools**:



2. Choose the **Save** icon to keep a copy of the generated `.W` file:

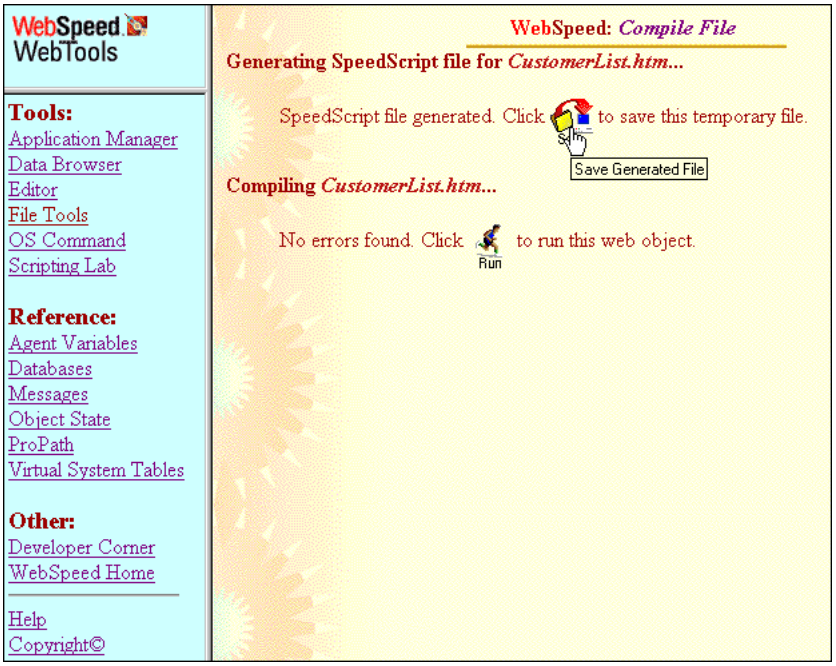


Figure 3–9 shows the Debugger working with the Embedded SpeedScript application.

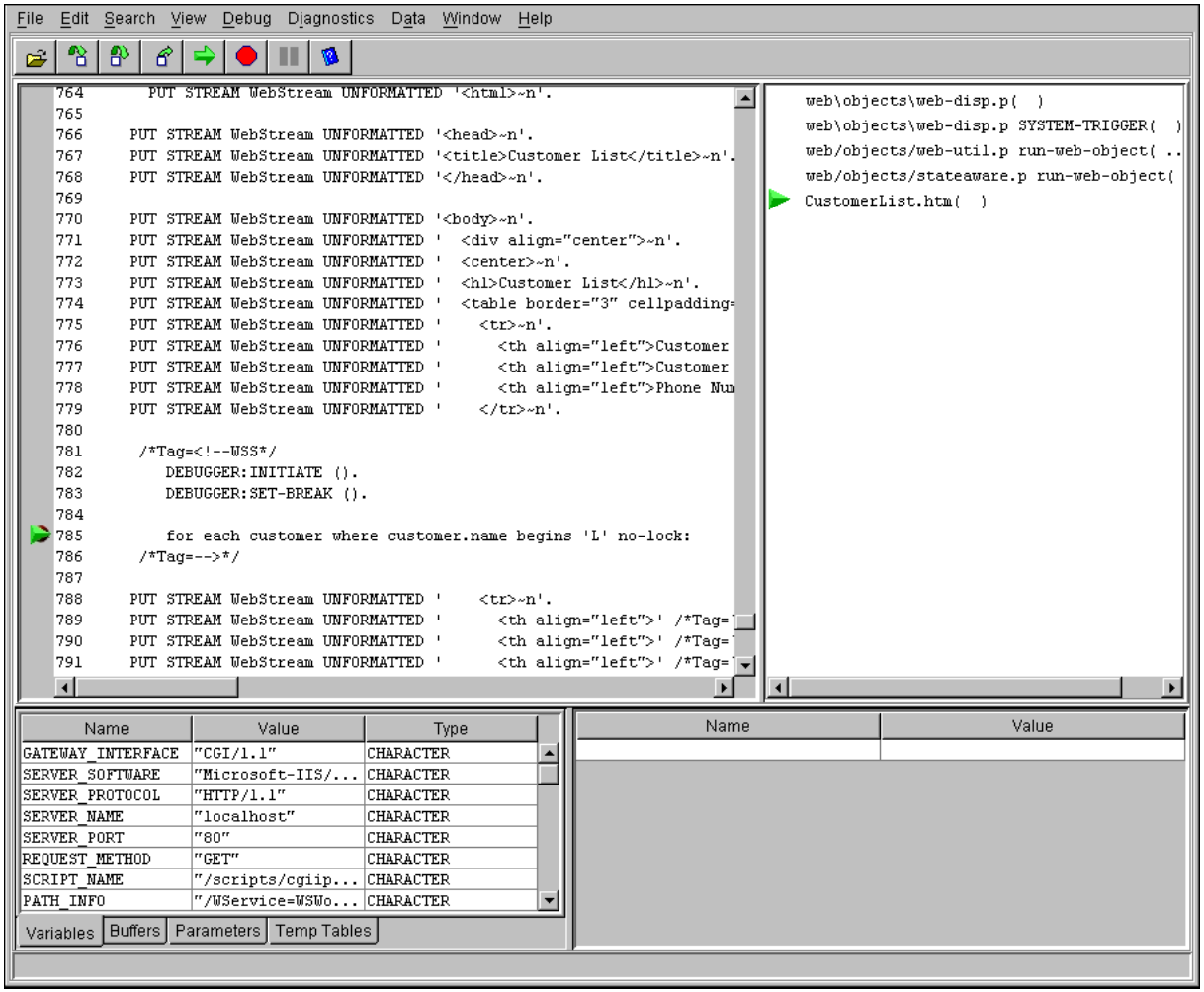


Figure 3–9: OpenEdge Application Debugger working in an Embedded SpeedScript application

The Debugger is available with some versions of UNIX. Linux is not currently supported. If you are using an X-Windows terminal and you want to use the Debugger with WebSpeed, you need to set a variable in the WebSpeed server's **Environment Variables** area, as shown in Figure 3–10. The `DISPLAY` setting is described in your UNIX documentation. `DISPLAY` refers to the X-Windows machine that will display graphical output. After you make this change, you must stop and restart the WebSpeed server.

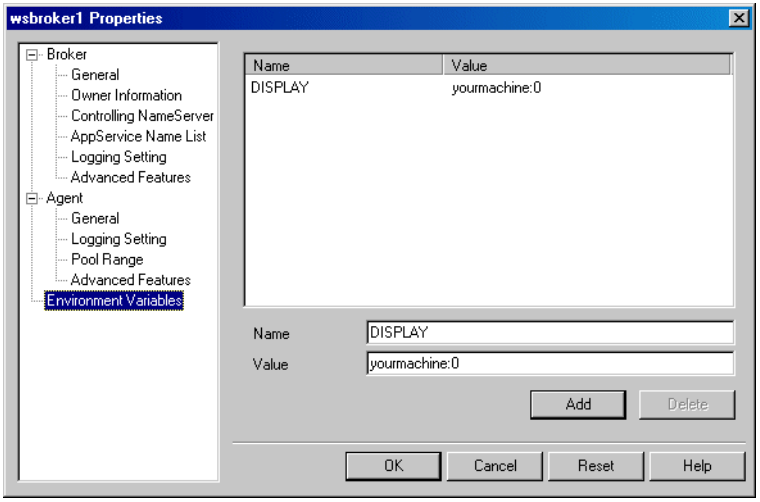


Figure 3–10: WebSpeed server environment variable: DISPLAY

If you cannot use the Progress Explorer to update the `DISPLAY` settings, then modify the `ubroker.properties` file and add or modify the following settings, as shown in Example 3–5.

Example 3–5: Setting DISPLAY environment variable in ubroker.properties

```
#
# WebSpeed Broker definition
#
[UBroker.WS.brokername]
    ...
    environment=brokername
    ...

#
# Environment for Broker: brokername
#
[Environment.brokername]
   DISPLAY=yourmachine:0
```

Using other techniques to debug WebSpeed applications

If you do not have access to the source code, then there are a few options you can use to see what is happening in the code. You can:

- Use the 4GL Trace option.
- Look in the agent log file.

The 4GL trace feature will log `RUN` statements, user-defined `FUNCTION` calls, and `PUBLISH` and `SUBSCRIBE` to the AppServer log file.

To enable the 4GL trace feature, you must manually modify the `ubroker.properties` file and add the `srvrLoggingLevel` and the `srvrLogEntries` parameters to the WebSpeed server's configuration. An excerpt of an entry is shown in Example 3–6. Stop and restart the WebSpeed server to start the logging.

Example 3–6: Enabling 4GL Trace in ubroker.properties

```
[UBroker.WS.DemoWS]
    srvrLoggingLevel=4
    srvrLogEntries=2
    srvrAppMode=Production
    brokerLogFile=C:\DemoApp\Logs\WS_broker.log
    srvrLogFile=C:\DemoApp\Logs\WS_agent.log
    srvrMinPort=5321
    srvrMaxPort=5325
    maxSrvrInstance=5
    controllingNameServer=NS1
    srvrStartupParam=-p web\objects\web-disp.p -weblogerror
    uuid=527a0623fe008210:67d940:f6484c1312:-7d87
    workDir=C:\DemoApp
```

The agent's log file contains warning and error messages, as well as the output from the 4GL MESSAGE statement. Make sure that the -weblogerror parameter is in the agent's startup parameters. Example 3–7 shows an excerpt from an agent's log file. It contains errors ("** Invalid character ..."), MESSAGE output ("MyAppMesg"), as well as the output from the 4GL trace.

Example 3–7: WebSpeed agent log file

```
WTA server initialising. (8835)
Connected to database sports2000, user number 9. (9543)
Run util/style.w PERSIST [Main Block - xxx.r]
Func checkAppMode "PRODUCTION" [Main Block - xxx.r]
Run process-web-request  [Main Block - xxx.r]
Run outContType in util/util.p "text/html" [process-web-request - xxx.r]
Run GetField in webutil/webstart.p "html " [process-web-request - xxx.r]
** Invalid character in numeric input x. (76)
Run showpage.r  [Main Block - showdetails.r]
** "showpage.r" was not found. (293)
Run destroyObject in webutil/ping.p [destroy - web2/admweb.p]
Run deleteOffsets in webutil/ping.p [destroyObject - web2/admweb.p]
Run SUPER  [destroyObject - web2/admweb.p]
MyAppMesg : Before Processing
Run doProc.p  [Main Block - main.r]
MyAppMesg : After Processing
```

The lines in the agent's log file are preceded with the agent number, the date and time, and the Operating System's Process ID of the agent. This makes it easy to determine which WebSpeed agent is running which code. Example 3–8 shows complete lines from the log file.

Example 3–8: WebSpeed agent log file lines

```
S-0001>(22-Jul-03 16:19:12:021) [124] MyAppMesg : Before Processing
S-0001>(22-Jul-03 16:19:12:021) [124] Run doProc.p  [Main Block - main.r]
S-0001>(22-Jul-03 16:19:13:821) [124] MyAppMesg : After Processing
```

Since the time is given, you can gain a rough estimate of how long each procedure took to run, if you surround the RUN command with MESSAGE statements. You can also use the {&FILE-NAME} and {&LINE-NUMBER} preprocessor variables to show which line of code the MESSAGE statement is on and which file it is in.

However, making the agent log all the RUN commands by using the 4GL trace option will cause the agent log file to grow quite quickly. You should enable this logging only when necessary.

Using the AppServer to access the business logic

Because WebSpeed agents are Progress 4GL clients, they can also use the AppServer to run the business logic for the application. This allows developers to share business logic with other Progress 4GL clients. This future-proofs your application's business logic.

When you design your application, all code that needs to access a database should be put into separate `.P` files to be (possibly) run on an AppServer. If you design your application this way, you can deploy your application with, or without, an AppServer very easily.

Example 3–9 shows code that will run on the client, and if there is an AppServer to process the logic, it will run it on the AppServer. If there is no AppServer, the code will run locally. This means that the same r-code files must be installed on the AppServer as well as on the client. The easiest way to do this is to use r-code libraries: one procedure library for the client r-code files and one for the business logic r-code files. You would then deploy both procedure libraries to the client, and only the business logic procedure library to the AppServer.

Example 3–9: Code designed to run on Client or AppServer

```
DEFINE VARIABLE hAppSrv    AS HANDLE  NO-UNDO.
DEFINE VARIABLE vConnected AS LOGICAL NO-UNDO.

CREATE SERVER hAppSrv.

vConnected = hAppSrv:CONNECT ("-AppService Sports2000_AS") NO-ERROR.
IF NOT vConnected
THEN
  DO:
    DELETE OBJECT hAppSrv.  /* Tidy up memory */
    hAppSrv = SESSION.      /* Make the hAppSrv handle "local" */
  END.

RUN foo.r ON hAppSrv.       /* Run the application */

IF vConnected               /* Release the connection and memory */
THEN
  DO:
    hAppSrv:DISCONNECT () NO-ERROR.
    DELETE OBJECT hAppSrv NO-ERROR.
  END.
```

If you don't want to use the AppServer to run your code but want to share your code with AppServer clients, you just need to put the business logic procedure library in the WebSpeed agent's PROPATH and use a normal run command.

For information on how to develop AppServer applications, see *OpenEdge Application Server: Developing AppServer Applications*. R-code libraries are documented in *OpenEdge Deployment: Managing 4GL Applications*.

Examples of simple and complex configurations

The simplest configuration, shown in Figure 3–11, is normally used by developers. It has all the components on one machine. This type of configuration is usually on a Windows platform that has all the tools needed to develop and test WebSpeed and AppServer applications. The Windows platforms supported by WebSpeed and AppServer in development mode are Windows NT or higher, not Windows 98 or ME.

Note: In Figure 3–11 and Figure 3–12, the shaded areas denote a physical machine.

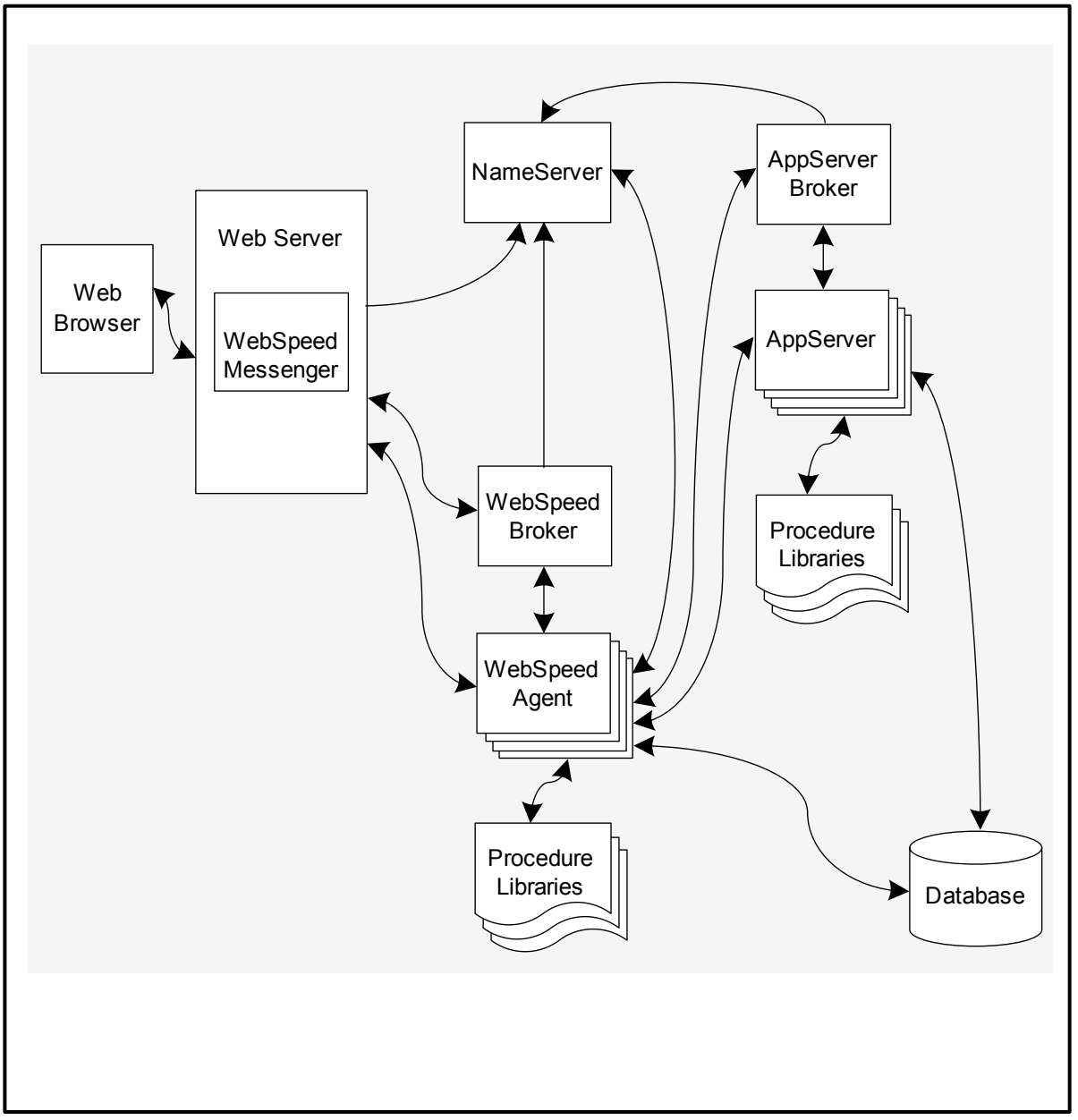


Figure 3–11: Simple configuration

Note: The components shown in Figure 3–11 are contained in OpenEdge Studio. You need to make sure you install the OpenEdge Development Server.

The most complex configuration, shown in Figure 3–12, is when you have all the components on separate machines and are using firewalls. In the complex example, the WebSpeed server is using the Internet NameServer to announce its availability, but the 4GL code that runs on the WebSpeed server uses the Intranet NameServer to find the appropriate AppServer.

In the complex configuration (Figure 3–12), the rightmost firewall is probably overkill. You would generally not use it in the real world, but it will make the deployment that tiny bit more secure.

The areas separated by the firewalls may be considered a “site.” This shows which components should be located together. The bandwidth required between the sites is much lower than the bandwidth needed within the sites. This is due to the fact that more information is moved between the components on one site, than between components on separate sites. See the “Request round-trip process” section on page 2–6 for details of the network traffic.

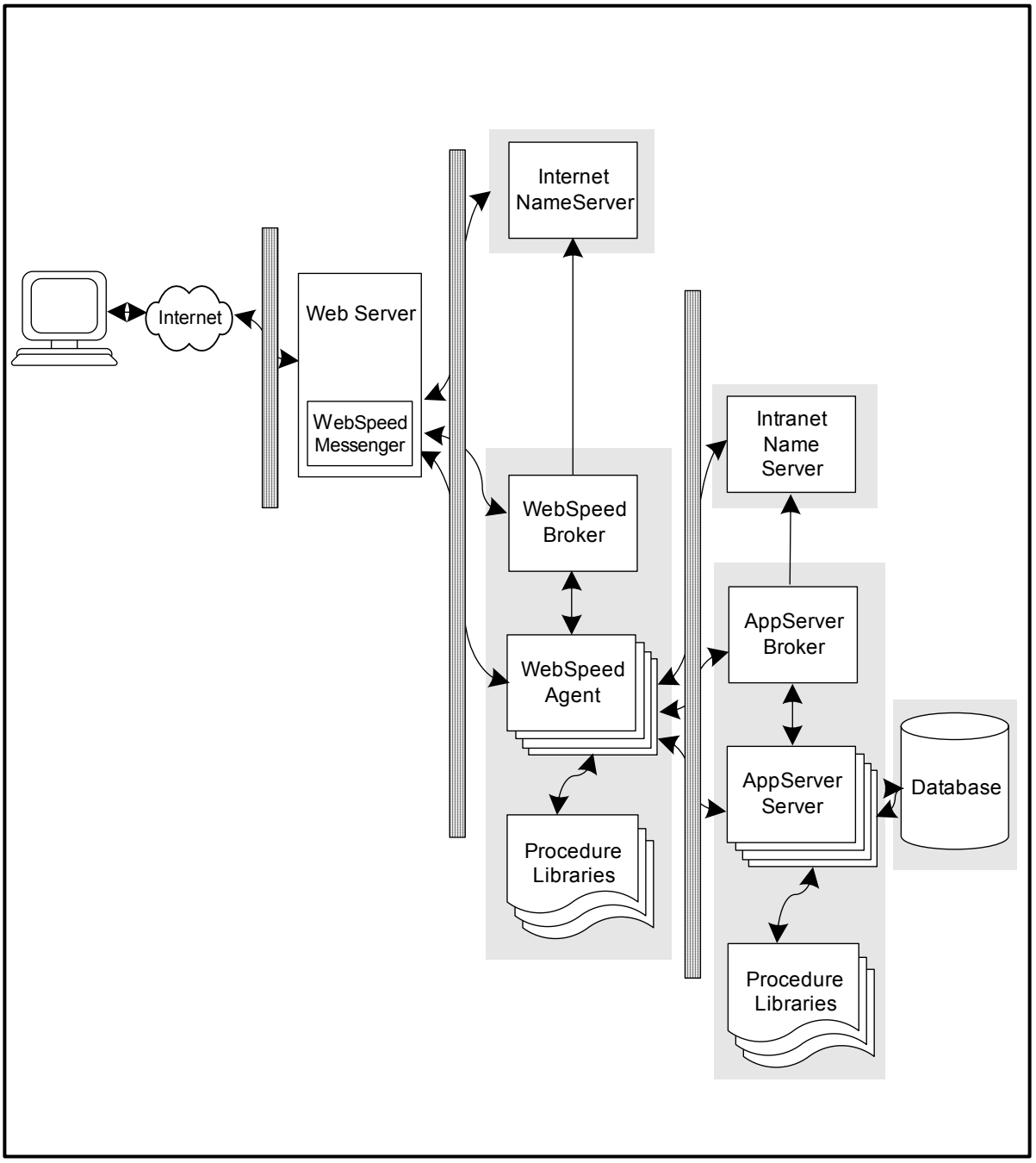


Figure 3–12: Complex configuration

4

Configuration of the AppServer

This chapter discusses how to configure an AppServer. It includes the following topics:

- General configuration
- AppServer security
- Error handling
- Debugging

General configuration

The AppServer is configured in a very similar way to the WebSpeed server. As in WebSpeed, it is good practice to keep the port range for the AppServers to the smallest possible range. If you are configuring an AppServer to have up to five servers, then the port range for the servers should be five.

Figure 4–1 shows an example of an AppServer configuration having only five servers. Note the **Minimum** and **Maximum port numbers**. If you set this port range too small or if another process uses a port in this range, then when the AppServer broker tries to launch an AppServer server, the server will try to use one of the currently in-use ports and fail to start.

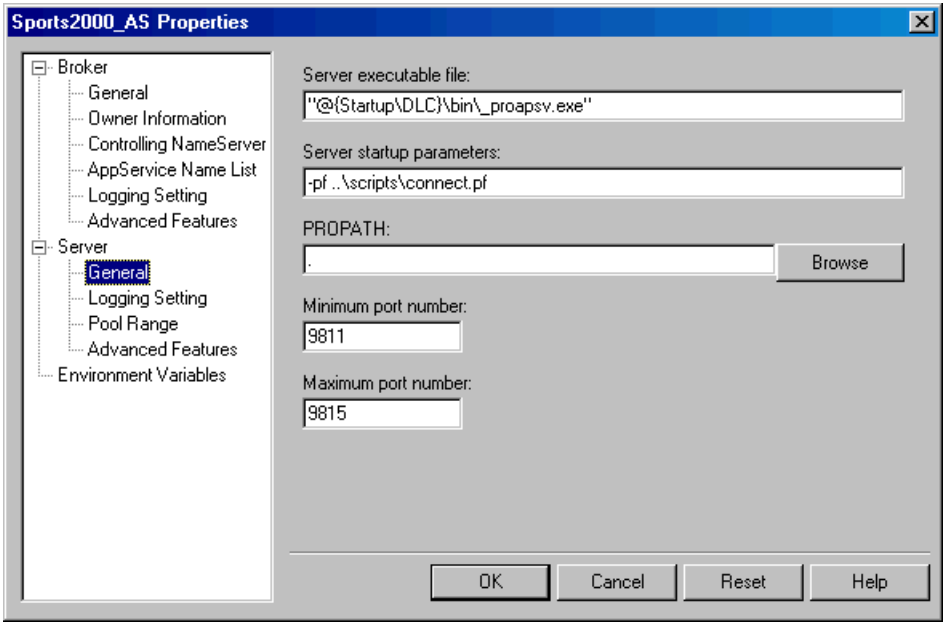


Figure 4–1: AppServer configuration with five agents

The AppServer broker and server logging settings, issues, and recommendations are the same as for WebSpeed, so the same comments and recommendations apply. See the "General WebSpeed server configuration" section on page 3–3 for details.

AppServer security

The measures taken to secure an AppServer application are equally as important as the measures to secure a publicly facing WebSpeed application. It is easier to control an AppServer application when there is no direct interaction with the end user. The AppServer broker's configuration should also specify an owner. This will allow the AppServer broker and servers to be started with the specified user's rights, not the root or system administrator's rights. This is covered in the "Broker ownership" section on page 2–4.

Controlling AppServer entry points

By default, the client process of an AppServer can run any code that exists on the PROPATH of the AppServer. To maximize security, you should limit the procedures that can be run. You can control AppServer entry points for an application server process at run time using the EXPORT method on the SESSION handle. AppServer entry points are the pathnames of procedures in the AppServer PROPATH that a client application can execute as remote procedures (persistently or nonpersistent). The EXPORT method establishes entry points by allowing you to set and maintain an export list that contains the pathnames of authorized remote procedures.

Setting the export list is usually done in the CONNECT procedure of the AppServer if you are using a state-aware or a state-reset AppServer configuration. If you are using a stateless AppServer configuration, the export list is typically set in the ACTIVATE procedure. The different states of an AppServer and the CONNECT/ACTIVATE procedures are discussed in the "AppServer configuration procedures" section on page 6–10.

To set the export list, the AppServer code passes a comma-separated list of procedures that the AppServer can access to the EXPORT method. The list can contain wild cards to make it easier to add entire directories or r-code Libraries. If you want to allow total access to all procedures, just call the EXPORT method without any parameters.

Example 4–1 shows a CONNECT procedure that implements an export list. More detailed information is in *OpenEdge Application Server: Developing AppServer Applications*.

Example 4–1: Connect procedure with export list

```
DEFINE INPUT PARAMETER pUserId        AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pPassword      AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pAppServerInfo AS CHARACTER NO-UNDO.

/* Authenticate user */
FIND FIRST AppUser WHERE AppUser.AppUserId = pUserId
                     AND AppUser.AppPasswd = ENCODE (pPassword)
                   NO-LOCK.
IF NOT AVAILABLE AppUser
THEN
  RETURN ERROR "Invalid UserId or Password".

/* Authorize access to particular procedures */
IF NOT SESSION:EXPORT (AppUser.ValidProcs)
THEN
  DO:
    /* If the EXPORT fails, log and refuse the connection */
    MESSAGE "Can't set export list for" AppUser.AppUserId.
    RETURN ERROR "Can't set export list".
  END.
```

Later in the application, if you want to clear the export list and set it to a different one, just invoke the EXPORT method again, as Example 4–2 shows.

Example 4–2: Resetting export list

```
.
.
.
/* Clear the export list to allow all procedures */
IF NOT SESSION:EXPORT ().
THEN
  DO:
    /* log failure and return */
    MESSAGE "Can't clear export list at {&LINE-NUMBER} in {&FILE-NAME}".
    RETURN ERROR "Can't clear export list".
  END.

/* Reset export list */
IF NOT SESSION:EXPORT ("accnts/*.r,common/qry*.r,system.pl<<*>>")
THEN
  DO:
    /* log failure and return */
    MESSAGE "Can't set export list at {&LINE-NUMBER} in {&FILE-NAME}".
    RETURN ERROR "Can't set export list".
  END.
.
.
.
```

Using DBAUTHKEY to lock your r-code to the database

An underused feature of OpenEdge is the DBAUTHKEY (and RCODEKEY) features of PROUTIL. With DBAUTHKEY, you assign a key to the database and then any code compiled against that database will have the key in it. When it is time to run the code, if the key in the database does not match the key in the r-code, then you will get an error similar to the following:

```
** CRC for table does not match CRC in program. Try recompiling. (1896)
```

If you already have r-code deployed, you can use the RCODEKEY function of PROUTIL to tag the existing r-code without having to recompile.

See *OpenEdge Data Management: Database Administration* for more information on using the DBAUTHKEY and RCODEKEY features of PROUTIL.

Error handling

An AppServer application is more easily controlled than a WebSpeed application. This is because the user does not interact with any application parameters being passed when using the AppServer. WebSpeed applications need to use URL parameters, cookies, or hidden fields to pass parameters, and these can be modified by the user.

As a matter of course, you should always check parameters for "sensibleness." This could be as simple as a range check or as complex as making sure records match a passed `ROWID`.

Making sure that errors do not occur is easier than trying to track them down after the event. Use the `MESSAGE` statement to track down errors that should not occur. If you are expecting to receive an integer between 1 and 9 and instead you get 15, then log this with a `MESSAGE` and return to the calling procedure by passing an `ERROR` on the `RETURN` statement.

Example 4–3 shows how to run a procedure and then check to see if an error was returned. It also shows how to pass back a message with the error.

Example 4–3: Error code

```
/* Calling Program */

RUN foo.p ON hAppSrv (15) NO-ERROR.

IF ERROR-STATUS:ERROR
THEN
  DO:
    /* Check for Progress errors first, like (293) */
    IF ERROR-STATUS:NUM-MESSAGES > 0
    THEN
      DO vLoop = 1 TO ERROR-STATUS:NUM-MESSAGES:
        MESSAGE ERROR-STATUS:GET-MESSAGE (vLoop)
                VIEW-AS ALERT-BOX ERROR.
      END.
    ELSE
      MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX ERROR.
  END.

------------------------------------------------------------

/* foo.p */

DEFINE INPUT PARAMETER pIn AS INTEGER NO-UNDO.

IF pIn < 1 OR pIn > 9
THEN
  RETURN ERROR "Invalid Parameter Passed".
```

A STOP condition on the AppServer causes the remote procedure request (persistent or nonpersistent) to terminate and will propagate the STOP condition to the client application. You can avoid this by handling the STOP condition on the AppServer with an ON STOP statement. A QUIT condition causes the remote procedure request to terminate and return successfully without any propagated condition to the client. However, the broker also terminates the client connection to the AppServer. Likewise, an ON QUIT statement can override this behavior. See *OpenEdge Application Server: Developing AppServer Applications* for more information.

Coding practices to avoid deadlocks

If you do not use good coding practices, there is a distinct possibility that you could lock yourself when accessing records in a database used by the client and the AppServer. Since the client and AppServer use two distinct connections, if you lock a record on the client and then try to access it on the AppServer, you will get a deadlock. It will appear that the AppServer process has hung. If two clients did the same thing, the following message would appear:

```
filename in use by user on tty.  Wait or choose CANCEL to stop. (2624)
```

Since there is no user interface when running code on an AppServer, this error cannot be seen or acted on by anyone.

The Lock Wait Timeout (-lkwtmo) parameter alleviates this issue. If the process tries for a lock and the time spent waiting exceeds the number of seconds specified in the -lkwtmo parameter, then the code will raise the STOP condition. This can have very bad side effects if you expect the record to be available after a find. To avoid this, you should always write code that will use NO-WAIT NO-ERROR and check to see if the record is available.

Example 4–4 and Example 4–5 show the before and after code.

Example 4–4: Problematic code without NO-WAIT NO-ERROR

```
DO TRANSACTION:
  FIND FIRST Customer WHERE Customer.CustNum = 3
                      EXCLUSIVE-LOCK.
  Customer.Balance = Customer.Balance + 20.00.
END.
```

Example 4–5: Robust code with NO-WAIT NO-ERROR

```
DO TRANSACTION:
  FIND FIRST Customer WHERE Customer.CustNum = 3
                      EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
  IF AVAILABLE Customer
  THEN
    Customer.Balance = Customer.Balance + 20.00.
  ELSE
    DO:
      IF LOCKED Customer
      THEN
        RETURN ERROR "Customer Locked".
      ELSE
        RETURN ERROR "Customer Not Found".
    END.
END.
```

Debugging

There are two main ways to debug an application. The first method involves accessing the source code for the application and the second method does not.

Using the OpenEdge Application Debugger to debug AppServer applications

The OpenEdge Application Debugger allows you to debug AppServer code from the client machine. If you start a debugging session on the GUI client and the code runs a procedure on the AppServer, the Debugger displays this code and allows you to control it.

To enable debugging, check the **4GL debugger enabled** check box in the **Advanced Features** for the AppServer, as shown Figure 4–2.

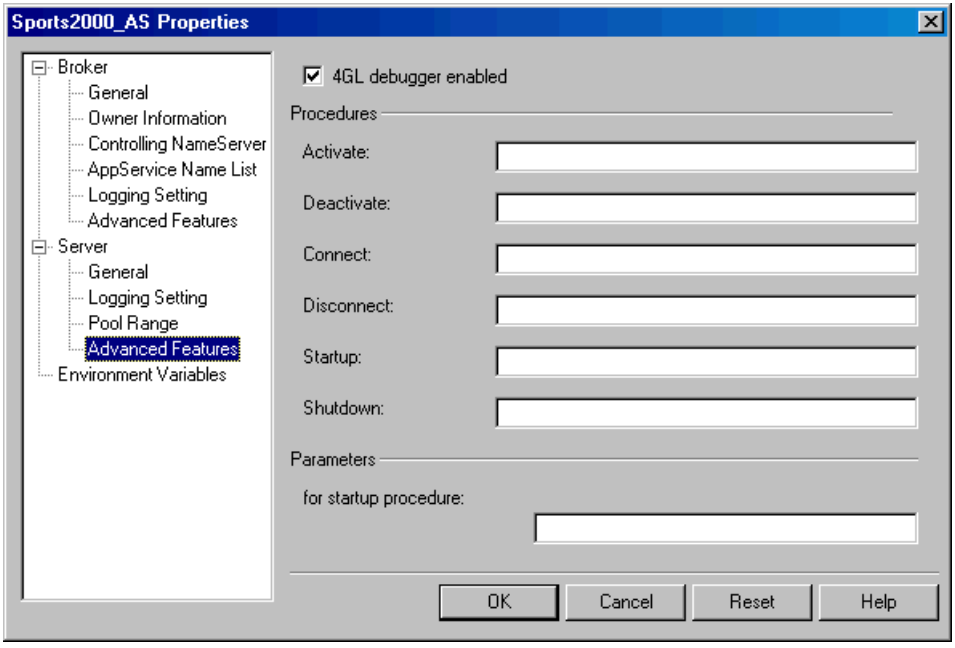


Figure 4–2: Enabling 4GL Debugger for AppServer server

Remote debugging

The `main.p` code in Example 4–6 shows an example of remote debugging. In this example, the `main.p` code is on a Windows GUI client, the AppServer code (`foo.p`) is in the `PROPATH` for the AppServer `Sports2000_AS`.

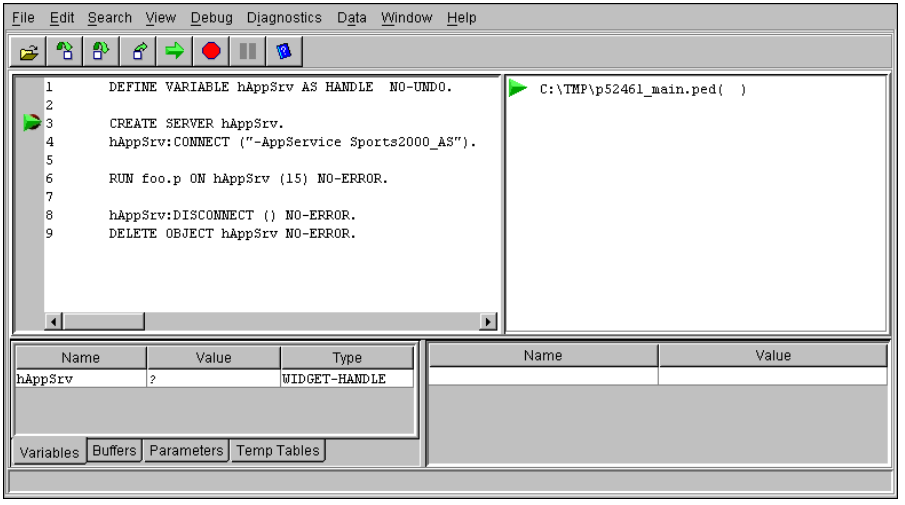
Example 4–6: Remote debugging code

```
/* main.p */
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.

CREATE SERVER hAppSrv.
hAppSrv:CONNECT ("-AppService Sports2000_AS").

RUN foo.p ON hAppSrv (15) NO-ERROR.

hAppSrv:DISCONNECT () NO-ERROR.
DELETE OBJECT hAppSrv NO-ERROR.

---------------------------------

/* foo.p */
DEFINE INPUT PARAMETER pIn AS INT NO-UNDO.

MESSAGE "{&FILE-NAME} Passed " pIn.
```

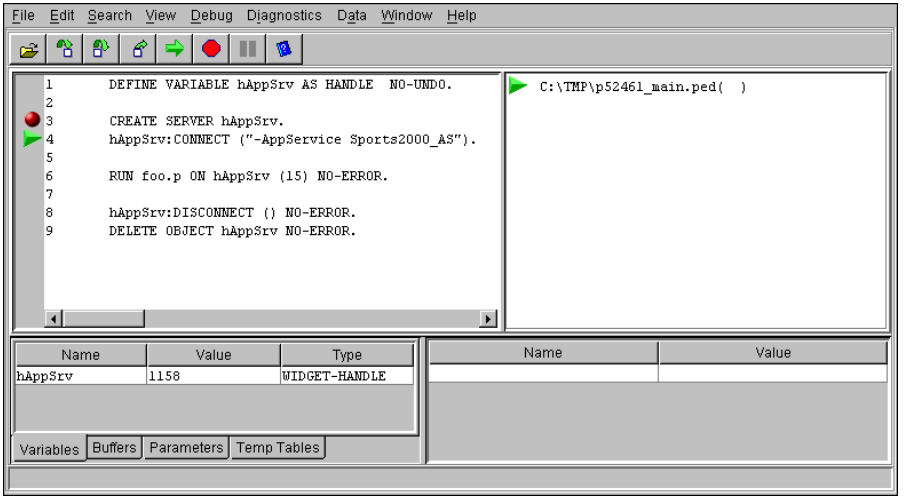
▶ To use the Debugger, from the Procedure Editor, choose the **Compile** menu option and then choose **Debug**.

The Debugger screens that follow show a sequence of events after every **Step Into [F7]** command:

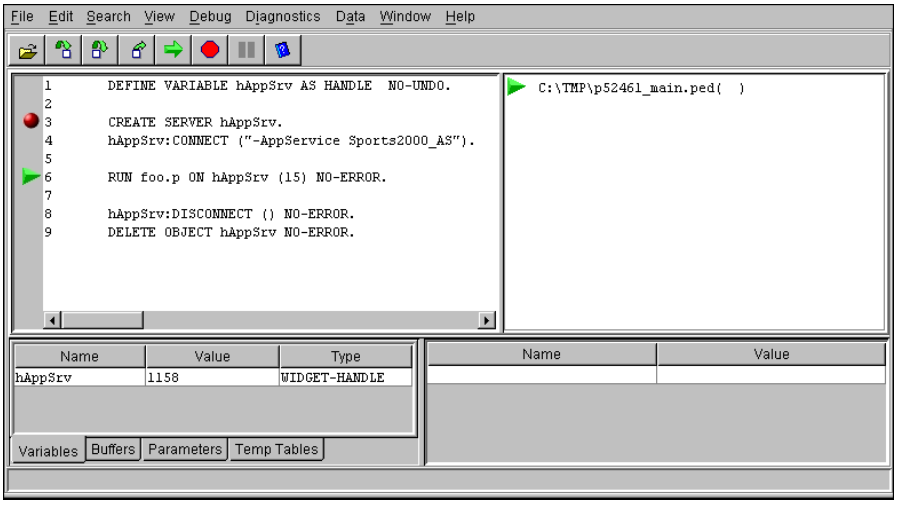
1. The following figure shows the initial state of the debugger. It has paused at the first executable line of code:



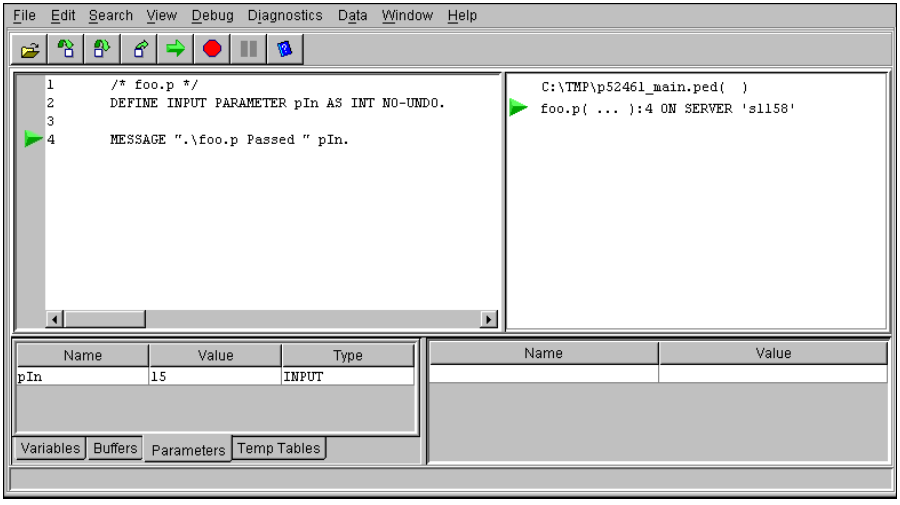
2. The following figure shows the Debugger after the first step:



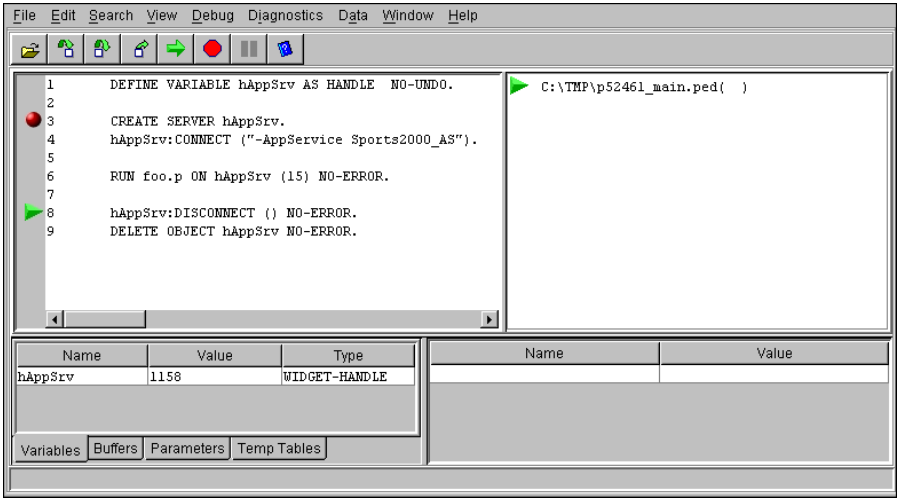
3. Next, you are connected to the AppServer and are about to run `foo.p`:



4. Next, you are in the code that is running on the AppServer. The right side of the window shows a procedure call stack. Note that the Server is **1158**, which is the server held in the `WIDGET-HANDLE` that points to **1158**. This `WIDGET-HANDLE` is `hAppSrv` on the client. The parameter(s) are shown on the tab folder in the lower left corner of the window.



5. Finally, the code has returned to the calling procedure, and it will now continue on to the next lines of code:



Using other techniques to debug AppServer applications

If you do not have access to the source code, then there are two options you can use to see what is happening in the code:

- Use the 4GL Trace option.
- Look in the server log file for errors.

The 4GL trace feature will log `RUN` statements, user-defined `FUNCTION` calls, and `PUBLISH` and `SUBSCRIBE` to the AppServer log file.

The 4GL trace feature will work for the GUI or Character client connected to an AppServer. To enable it for a GUI or Character client, add the following command line options. In this example, the log file will be in the working directory and be called `mylog.lg`:

```
-clientlog mylog.lg -logginglevel 4 -logentrytypes 2
```

To enable the 4GL trace feature for the AppServer, you must manually modify the `ubroker.properties` file and add the `srvrLoggingLevel` and the `srvrLogEntries` parameters to the AppServer's configuration. An excerpt of an entry is shown in Example 4–7. Stop and restart the AppServer to start the logging.

Example 4–7: Enabling 4GL Trace in ubroker.properties

```
[UBroker.AS.DemoAppSrv]
    srvrLoggingLevel=4
    srvrLogEntries=2
    operatingMode=State-aware
    brokerLogFile=C:\DemoApp\Logs\AS_broker.log
    srvrLogFile=C:\DemoApp\Logs\AS_server.log
    srvrMinPort=5521
    srvrMaxPort=5525
    maxSrvrInstance=5
    controllingNameServer=NS1
    PROPATH=.
    uuid=527a0623fe008210:67d940:f6484c1312:-7d87
    workDir=C:\DemoApp
```

If you use a `MESSAGE` statement in AppServer code, then the message will appear in the AppServer's log file. The following line of code in a file called `foo.p` in the `AS` directory when run on the AppServer will put the line after it in the log file. The preprocessor parameters show the current filename and line number within that file:

```
MESSAGE "{&FILE-NAME} {&LINE-NUMBER} Before Command".

--------------------

[03/07/23@22:10:54.991+0100] P-002592 T-002596 0 AS -- .\AS\foo.p 1 Before
Command
```

The AppServer log file fields are:

- Date and Time with offset from GMT.
- Process ID of the AppServer server (`P-002592`).
- The Java Thread ID.
- The Severity of the message. If you use the 4GL `MESSAGE` statement, the Severity is 0.
- The AS shows that this is AppServer.
- Two hyphens.
- The message text.

5

Configuration of the NameServer

This chapter discusses how to configure the NameServer. It contains the following topics:

Understanding the NameServer

The NameServer is an optional but integral part of the OpenEdge platform. The ability to provide location independence, load balancing, and connection-time fail-over is provided by the NameServer service. It can also be a single point of contention or failure if not configured appropriately.

When the NameServer is being used by the OpenEdge servers, they will inform the NameServer, by default every 30 seconds, that they are still alive and able to process requests. The **Registration retry** entry on the **OpenEdge Server Broker Advanced Features** page sets this time period.

If the NameServer does not receive the broker's message within the NameServer's `brokerKeepAliveTimeout` period, the broker is removed from the **Available Broker** list within the NameServer. To be usable, set the OpenEdge server's **Registration retry** setting to be equal to or less than the NameServer's `brokerKeepAliveTimeout`. The `brokerKeepAliveTimeout` is, by default, set to 30 seconds. To alter this, you will need to edit the `ubroker.properties` file manually.

If the broker is removed from the **Available Broker** list, the NameServer will no longer tell clients to connect to this broker, thereby providing connection-time fail-over.

The **Advanced Features** page of an AppServer broker is shown in Figure 5–1. The WebSpeed broker page is identical.

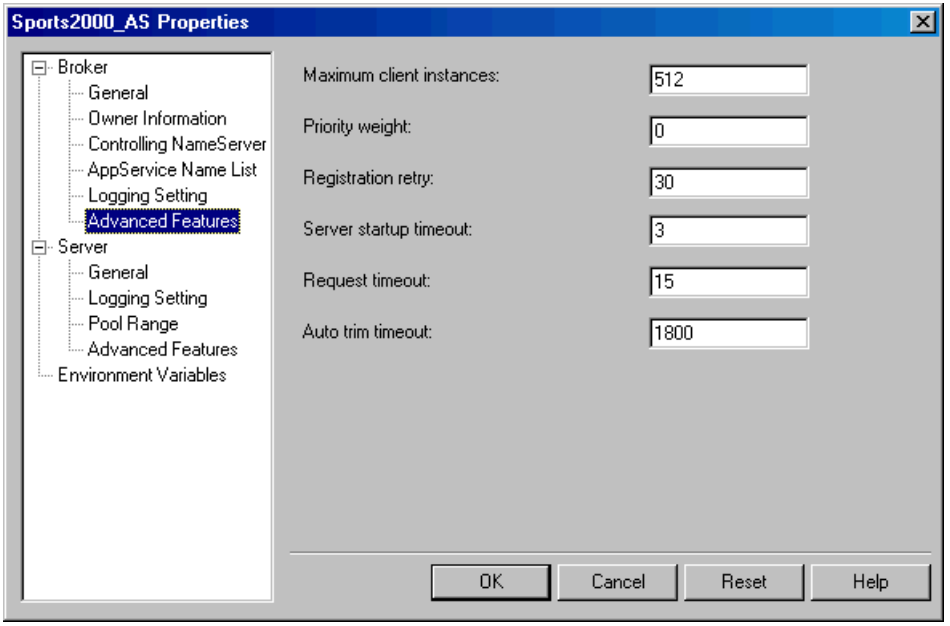


Figure 5–1: AppServer broker Advanced Features page

Location independence

Because the OpenEdge clients send a message to the NameServer requesting the address of a broker to handle their application request, it means that the host machine for the OpenEdge server can be changed without affecting the application source code.

Conversely, if you do not use a NameServer, you will need to change the client deployment, either source code or configuration file to point to the new broker's address. Using a "No NameServer" deployment is covered in the "No NameServer version of the request round-trip" section on page 2–8.

Load-balancing

So far, all of the examples show a single OpenEdge server servicing the client's requests. If this server becomes overwhelmed with requests, performance will drop. The solution to this is to allow load-balancing.

To provide load-balancing, you configure two (or more) OpenEdge servers that would be set up with identical Progress 4GL application code and database connections. These would normally reside on separate host machines for performance reasons.

Each of the OpenEdge servers would have its own unique name, but would also be configured to respond to a common service name. This is done in the **AppService Name List** page of the broker configuration. Figure 5–2 shows an AppServer broker that will respond to its own name, as well as the common name **Accounts**.

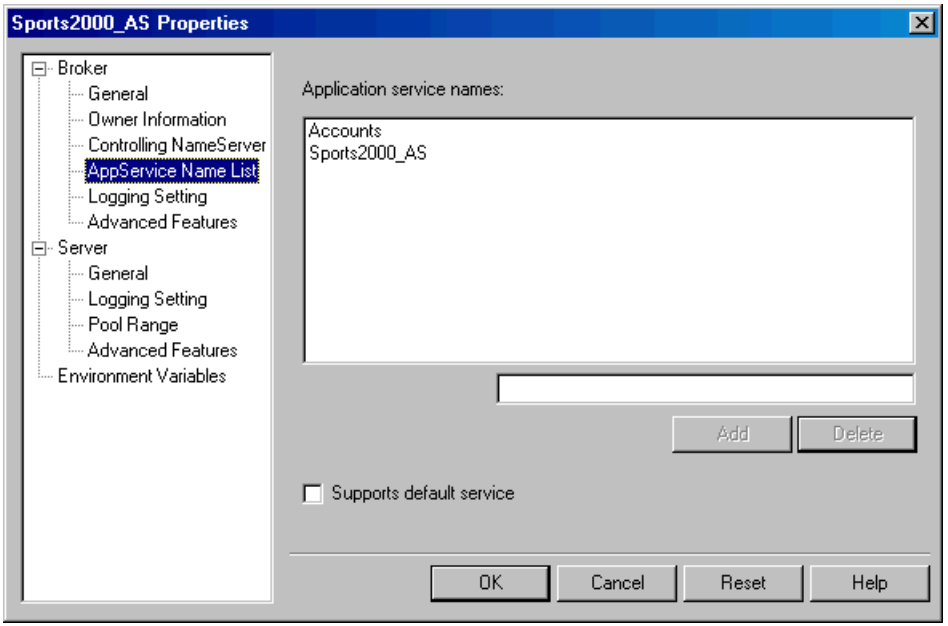


Figure 5–2: Setting application service names

Setting more than one application service name for a broker allows the NameServer to forward requests for all the listed services to that OpenEdge server.

You might want to allocate priority for each OpenEdge server to enable a larger machine with more WebSpeed agents or AppServer servers to be given more requests than a smaller machine. To do this, set the **Priority weight** on the **Advanced Features** page for the OpenEdge broker, as shown in Figure 5–3.

The number of requests sent to each broker is decided by a pro-rata formula. The percentage of all requests sent to a broker is calculated by taking the priority weight of the broker, dividing it by the total of all the priority weights for brokers able to respond to the request, and multiplying by 100.

Table 5–1 shows an example of three brokers with differing priority weights and the associated percentage of requests they will each service.

Table 5–1: Broker priority weights

Broker name	Priority weight	Percentage requests
Broker 1	10	18.2%
Broker 2	15	27.3%
Broker 3	30	54.5%

The NameServer will adjust these percentages each time a new broker joins or leaves the pool of brokers able to respond to the same request.

Figure 5–3 shows the priority weight being set on the Sports2000_AS AppServer.

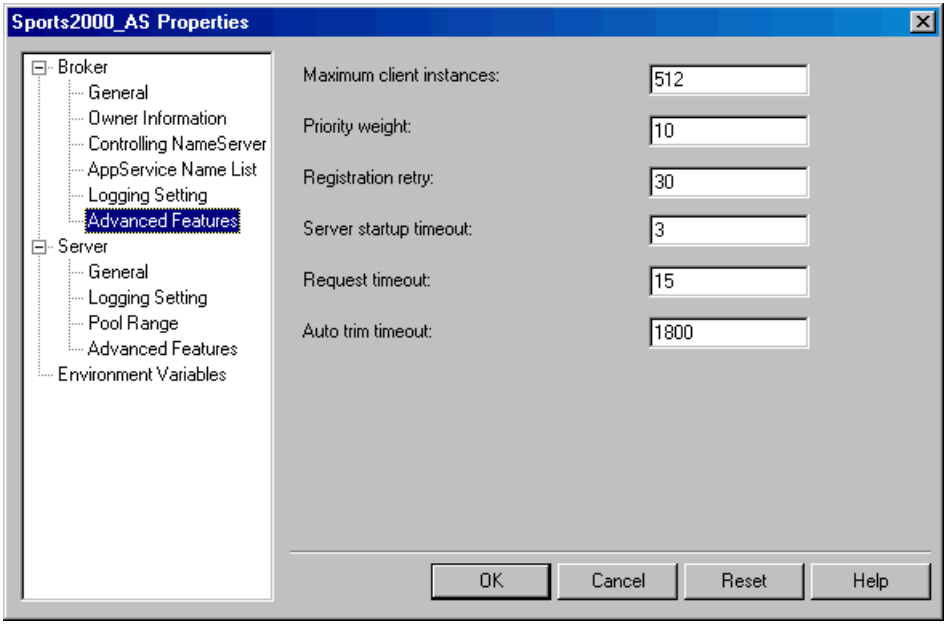


Figure 5–3: Setting Priority weight for an AppServer

This functionality is available only if you have licensed a 50- or 250-agent WebSpeed Transaction Server or if you have licensed the Load Balancing NameServer.

Fault-tolerant NameServer configurations

Having only one NameServer on a network that is responsible for allocating brokers to handle requests can cause performance issues and also becomes a single point of failure. These issues can be solved by setting up multiple NameServers using one of both of the following configurations:

- NameServer replication
- NameServer neighborhoods

For more information on setting up fault-tolerant NameServers, see *OpenEdge Getting Started: Installation and Configuration for Unix* or *OpenEdge Getting Started: Installation and Configuration for Windows*.

NameServer replication

Because the protocol used by the NameServer is UDP, it allows broadcast messages to be sent to all machines on the same subnet. Setting up two or more machines on the same subnet and having a NameServer on them provides for resilience using NameServer replication.

To configure this, put the NameServers on different machines listening on the same UDP port and the client and OpenEdge servers broadcasting the requests to the subnet. Figure 5–4 illustrates this configuration.

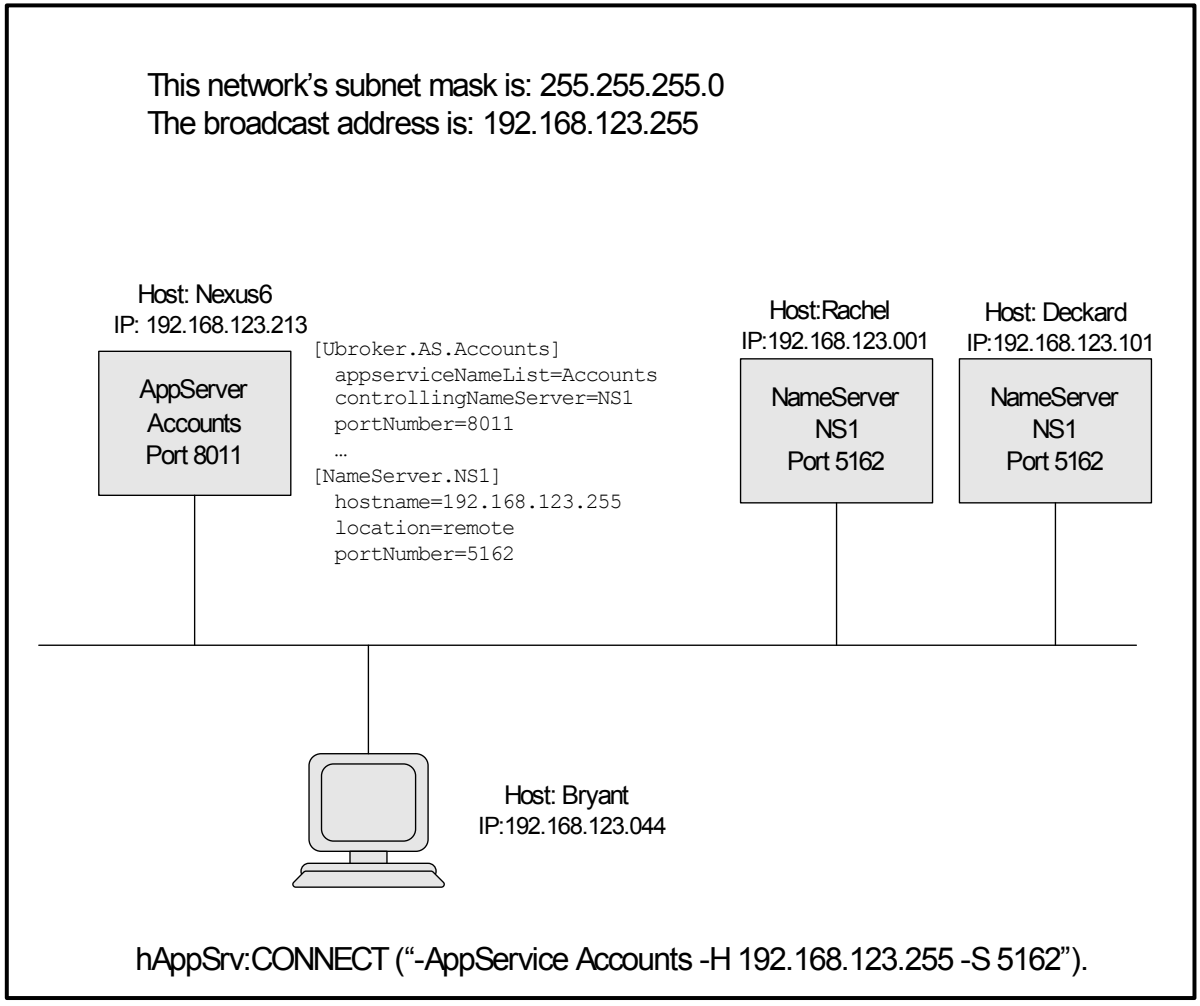


Figure 5–4: NameServer replication

In this example, the NameServer hosts are `Rachel` and `Deckard`, and each has a NameServer configured and listening on UDP port 5162. The AppServer is on `Nexus6` and is listening on TCP port 8011. The AppServer will send its registration message to the network subnet broadcast address 192.168.123.255 on UDP port 5162. Any NameServer on this subnet listening on port 5162 will receive this message and register this broker.

When the client makes a NameServer request to find a broker capable of servicing accounts requests, it also broadcasts to 192.168.123.255. All the NameServers will receive the request and respond. The client will just take the first positive response it receives. The others will be ignored.

Another benefit of using a broadcast message is that if you decide to move `Deckard` from 192.168.123.101 to 192.168.123.131 and stay within the same subnet, then you do not need to change any configuration on either the AppServer or the client code.

To make the AppServer broker broadcast its registration messages to the subnet, you need to configure the appropriate NameServer in the Progress Explorer, as shown in Figure 5–5, or edit `ubroker.properties` manually and enter the settings, as shown in Figure 5–4.

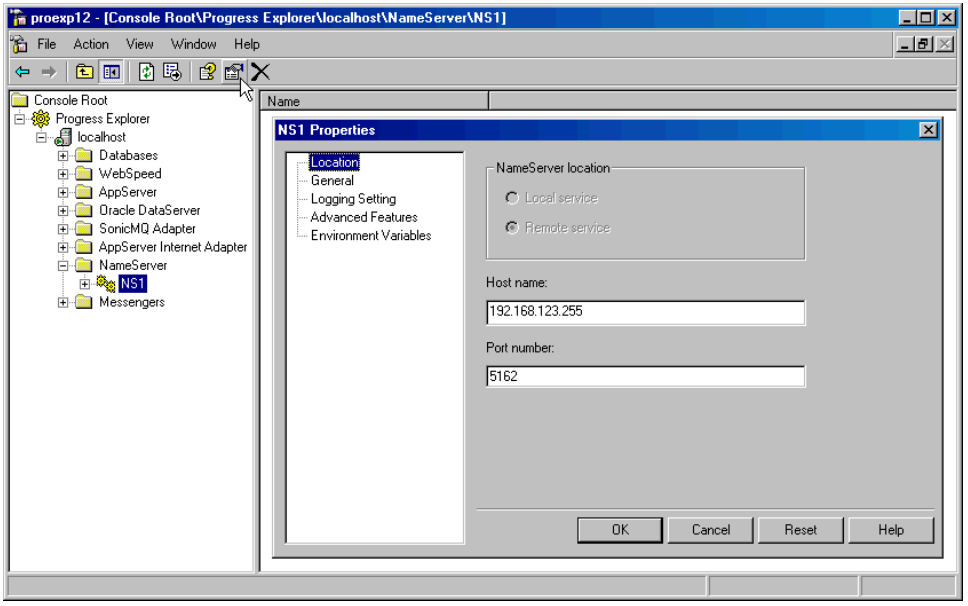


Figure 5–5: Setting NameServer broadcast properties

NameServer neighborhoods

NameServer neighbors are alternate NameServers that you specify as part of a NameServer configuration. When a NameServer receives a client connection request that it cannot resolve, it automatically passes the request to the specified NameServer neighbors to attempt the resolution.

To achieve this, add the neighboring NameServers to the `ubroker.properties` file as remote NameServers by using the Progress Explorer or a text editor. Then, add the list of neighboring NameServers to the primary NameServer's configuration.

For example, Figure 5–6 shows three NameServers in different subnets. The NameServer on `Roy` is neighbored to the other two NameServers. Note that the configuration file on `Roy` does not use the correct names for the other two NameServers. It does not need to. As long as the hostname and port number are correct, the name is irrelevant. In this way, all the NameServers are NS1, but there is no confusion. The AppServers in each subnet only have their local NameServer configured. The client also only connects to its local NameServer, but will get allocated any of the AppServers. This allocation is a result of the NameServers passing the request on if they cannot respond locally.

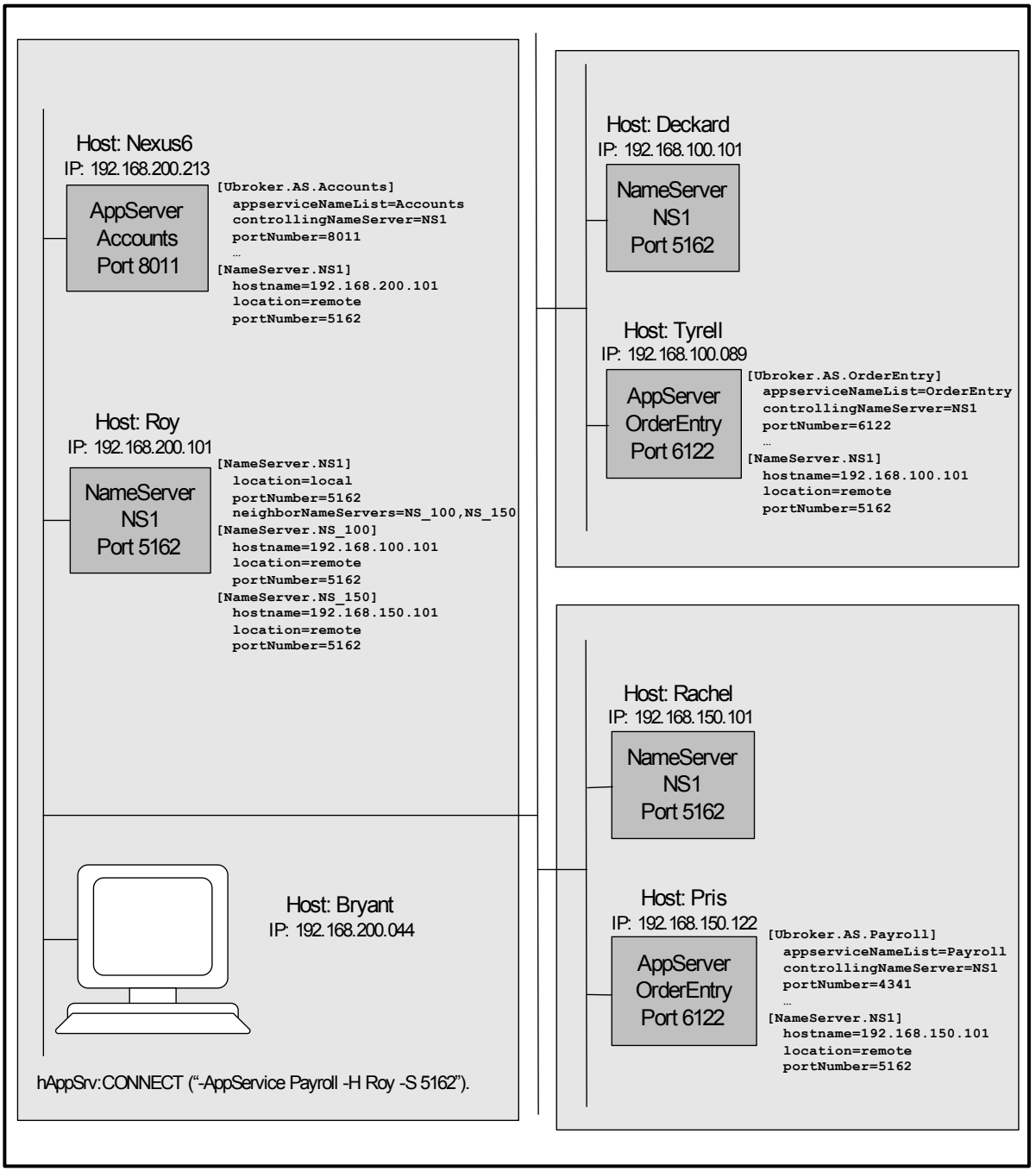


Figure 5–6: NameServer neighborhood

Figure 5–7 shows how to configure NS1 using the Progress Explorer. Example 5–1 shows the matching `ubroker.properties` file.

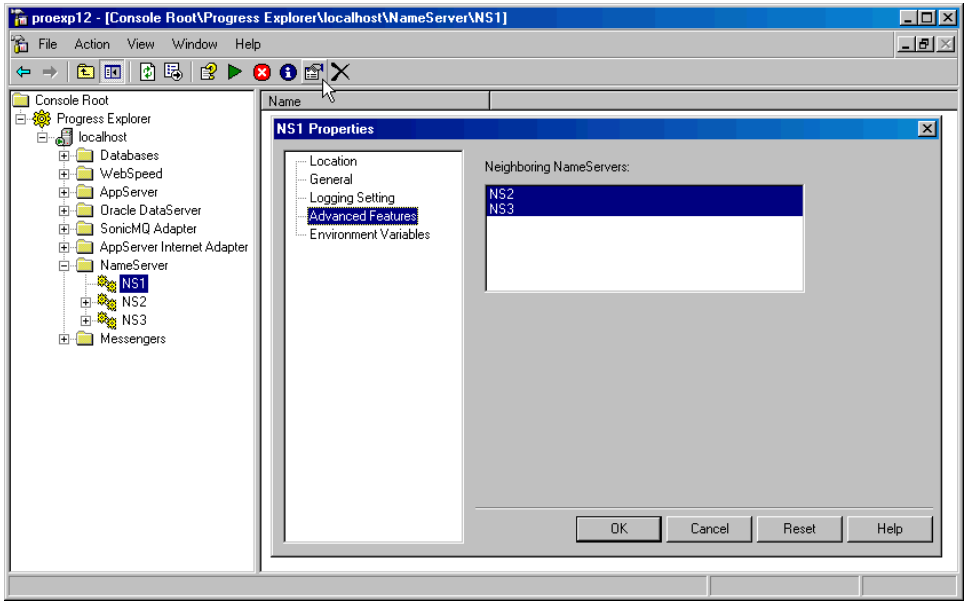


Figure 5–7: Setting NameServer neighbors in the Progress Explorer

Note: In the Progress Explorer, you need to highlight which NameServers you want to set as the local NameServer's neighbors.

Example 5–1: Setting NameServer neighbors in ubroker.properties

```
[NameServer.NS1]
  autoStart=1
  location=local
  portNumber=5162
  neighborNameServers=NS2, NS3
  srvrLogFile=@{WorkPath}\NS1.ns.log
[NameServer.NS2]
  hostName=Deckard
  portNumber=5162
  location=remote
[NameServer.NS3]
  hostName=Rachael
  portNumber=5162
  location=remote
```

Logging levels

There are three different logging settings for the NameServer: Error Only, Terse, and Verbose. The default is Terse, which provides enough information to be practical. You should leave the setting at Terse unless you need to debug access to OpenEdge servers. Earlier in this manual we discussed how to configure a firewall-enabled deployment, where Verbose logging is used.

Log file maintenance

Like the OpenEdge servers, the NameServer also has a log file that will grow over time. The same comments and recommendations apply equally to the NameServer. See the "General configuration" section on page 4–2 for details on maintaining the NameServer log file.

6

Performance Considerations

This chapter discusses the following topics:

- NameServer performance
- WebSpeed performance
- AppServer performance
- AppServer configuration procedures
- Application performance tuning

Note: For information on database performance tuning using Fathom, see *OpenEdge Revealed: Mastering the OpenEdge Database with Fathom Management.*

NameServer performance

The NameServer adds a small amount of time to each request made. This amount of time is very small and should not impact performance. There are other parts of the entire request that will cause more performance degradation than the NameServer. The NameServer provides many useful functions and unless it is really going to cause a problem, you should use the NameServer.

However, there is one feature of the NameServer that might cause some overhead: Broadcast UDP. This is discussed in the "NameServer replication" section on page 5–6. If you use the broadcast method of communications, then all the hosts on the subnet will receive the message and must handle it. This can cause slow or overworked hosts to become even slower. Do not use Broadcast UDP unless it is essential for redundancy or if you have a small number of machines on the subnet.

WebSpeed performance

This section discusses the following WebSpeed performance issues:

- How requests affect performance
- Browser (HTTP) response times
- HTTP/S performance
- Using different Messengers
- Multiple Web servers

How requests affect performance

If you recall in the "Request round-trip process" section on page 2–6, there are quite a few steps that make up the entire round-trip process and possibly quite a few separate machines, ranging from the Web server, firewalls, NameServer, WebSpeed server (broker and agents) and probably a database as well. Each of these steps introduces performance challenges of their own. The Web server must cope with not only the WebSpeed requests, but the normal HTML requests as well. Firewalls can introduce network latency, as some will inspect each packet to make sure it is "allowed" before passing it through to the next machine in the process. The NameServer performance issues are covered in the previous section.

The WebSpeed broker and agents have a role to play in performance as well. The broker launches and configures new WebSpeed agents before they are needed. This enables the requests that are being received to wait as short a time as possible in the broker's request queue. The launching of a new agent will take a period of time—the agent itself needs to be loaded into memory, possibly run some application code to create super procedures, and connect itself to the database. To keep free agents, you should set the **Minimum agents** to a number higher than 0. This setting controls how many agents the broker will keep free, up to the M**aximum agents**.

As Figure 6–1 shows, the broker will start five agents as soon as it is started. It will keep at least two free agents at all times unless it has already launched the maximum number of agents, which is 10. Agents that are not used for a period of time will be killed. The **Auto Trim Timeout** setting in the broker's **Advanced Features** tab controls this time period and is entered as a number of seconds, so the default of 1800 is equivalent to 30 minutes.

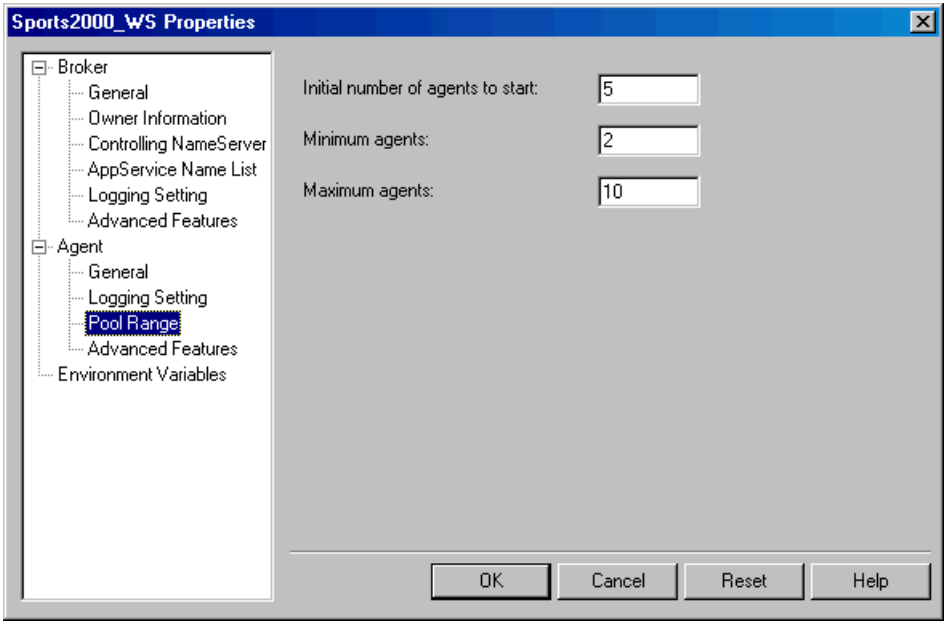


Figure 6–1: Setting minimum and maximum agents

Browser (HTTP) response times

Having a good response time for a Web site is very important. Making sure that the Web server is configured well and has enough memory and CPU performance is important in providing good response times.

Hosting static images and HTML on the Web server allows the Web server to cache these and provide better performance.

Using the HTTP network monitoring feature of Fathom Management lets you to track the performance (average response times) of your Web site. See the *Resource Monitoring Guide* for more information.

HTTP/S performance

Using HTTP/S to encrypt the traffic between the Web browser and the Web server provides very good security for the data, but it also introduces a performance hit. In general, for each request that is made to the Web server, the Web browser and Web server must go through at least 8 and up to 13 handshake messages before the actual data is sent. Also, one of these handshake messages needs the Web browser to generate a long random number, which is a slow process.

Because a Web page is usually made up of multiple requests, using HTTP/S as the protocol slows down the Web page being displayed. For example, a Web page with 15 images on it will mean 16 individual requests to be made to the Web server.

HTTP/1.1 has features that should allow one connection with multiple requests to work, but the implementation into the Web servers and the Web browsers has not been completed. There are hardware SSL accelerators on the market that will alleviate most of the performance issues on the Web server side when using SSL.

At present, Fathom Management cannot use HTTP/S as a protocol for the HTTP monitoring feature.

Using different Messengers

Throughout this manual, the WebSpeed Messenger that has been described is `cgiip` or `cgiip.exe`. Depending on your Web server, there are alternate WebSpeed Messengers. If you are using a Microsoft IIS Web server, you can use the `wsisa.dll` Messenger, and if you are using the Netscape/iPlanet Web server, which is now part of SunOne, you can use the `nsapi.dll`.

Each of these Messengers acts in exactly the same way as `cgiip`, but because they are Dynamic Link Libraries (DLL's), they stay in memory and are faster to execute the next time they are called.

Being a DLL does have a drawback. If the Web server gets confused, there is a strong possibility that the Messenger process will stop working. The only way to correct this is to restart the Web server process on the Web server machine; sometimes this involves a reboot.

Due to the `cgiip` Messenger being loaded each time a request is made, it is slower than the DLL versions, but it is also more reliable. Since the time it takes the `cgiip` Messenger to load itself into memory is quite small, using `cgiip` is a good idea for production Web sites, as the performance overhead is slight, but the reliability is high. You should test each possible Messenger for performance and determine which one you want to use. During testing, remember to time the entire application, not just the Messenger load times.

Multiple Web servers

One way to increase the throughput of the Web server is to have more than one and share the load. This is easily achieved using WebSpeed, because the Messenger configuration is identical on each Web server. To make more than one Web server respond to requests for the same Web site, you can use DNS round-robin aliases or a hardware redirection. For more information see your router or DNS documentation.

AppServer performance

This section discusses the following AppServer performance topics:

- How AppServer operating modes affect performance.
- Using asynchronous AppServer calls.

How AppServer operating modes affect performance

In the "Request round-trip process" section on page 2–6, the request/response round-trip was described. The AppServer operating mode has a bearing on performance and network traffic. The AppServer operating mode is determined at application design-time. The operating mode cannot be changed without rewriting the application so it can to use different modes. For example, you cannot just change the operating mode from state-aware to stateless unless the application has been designed to work in both modes.

Understanding state-reset, state-aware, and stateless operating modes

State-reset and state-aware are similar operating modes. They both preserve the AppServer's state between client requests. This is achieved by allocating one AppServer to each client connection. This allows the client to keep transactions open, have queries maintained, and have temp tables and variables available over multiple AppServer requests. The difference between the two modes is that when the client disconnects, a state-reset AppServer will return to the state it was in when it was first started—the same databases will be connected (if you disconnected or connected others), the same persistent-procedures will be started, and so on. In state-aware mode, the AppServer is left as is. This can be a good thing because records you have added to temp tables remain for the next client. This allows you to cache records for reuse.

Stateless AppServers are a totally different scenario. When the client connects to the AppServer with the `CONNECT` method, it only connects to the broker and is allocated a Client ID. Every time a `RUN ON SERVER` command is performed, the client connects to the broker to be allocated a server. When the request is finalized, the client disconnects from the server, which then becomes available for other clients to use.

To retain state information between client requests, the developer can use the `SERVER-CONNECTION-CONTEXT` attribute on the `SESSION` handle in the AppServer server to keep character information or use the `SERVER-CONNECTION-ID` attribute on the `SESSION` handle in the AppServer as a key into a database "state" table. These techniques are discussed in the "AppServer configuration procedures" section on page 6–10.

The fastest operating mode for an AppServer is state-aware. This is because it does not need to tidy up after the client disconnects, and it is always allocated to the AppServer client. Therefore, you do not have the overhead of the client asking the broker for a server which occurs with stateless. The second fastest mode is state-reset, because it needs to reset itself to the initial state it was in when it was first started — the same SESSION attributes, the same databases connected, and so on. The slowest mode is stateless because it has a large amount of communications overhead and usually more application code to keep state when needed.

If you are making several related requests very close to each other, you can increase the performance of a stateless AppServer by making the server bound. When you are bound to an AppServer, you are allocating the server to be accessed only by the one client in the same manner as state-aware until you become unbound. You can bind the client to the server by setting the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle in the AppServer to TRUE.

Using stateless mode

Generally, you should always design the AppServer application as deployable in stateless mode, but deploy the application in a state-aware mode for performance. So, when should you use stateless mode AppServers? The answer is when the machine hosting the AppServer becomes resource-bound and slows down. This will usually occur when the machine runs out of real memory and starts to use virtual memory. When you have a large number of AppServer clients connected to a state-aware or state-reset AppServer, there will be one server for each client, so, for example, 500 AppServer clients means 500 servers in memory. Each AppServer can use a large amount of memory depending on your application.

When you use stateless mode, you will usually need much fewer servers to cope with the same number of clients. This is because most of the time the AppServer is not doing anything, and in stateless mode it can be used by another client during this slack time. Having fewer servers running results in a proportionate saving in memory use.

By using stateless AppServers to alleviate the memory load on the machine, it will start to perform better, and the overhead of using stateless AppServers becomes much less than using a slow machine. The number of database connections will also decrease when you move to a stateless AppServer environment, thereby making the database machine less loaded as well.

You can monitor your system memory usage with Fathom Management’s System Resource monitor. You can monitor the amount of memory consumed by the AppServer using Fathom Management’s OpenEdge Server Resource monitor.

Using asynchronous AppServer calls

Another way to achieve perceived performance gains when accessing the AppServer is to use asynchronous AppServer calls. The application code on the AppServer does not need to be altered at all; this is purely a client-side programming construct.

Synchronous requests (the default) occur when the client blocks and waits for the result of the remote request before continuing execution.

Asynchronous requests occur when the client continues execution after initiating the request and processes the result whenever the AppServer makes it available. See the "Example of asynchronous requests" section on page 6–8 for when you might want to use this type of call.

Example of asynchronous requests

If you are hosting a Web site that needs to gather live information on the quantity of an item in each of a number of warehouses, you might develop an architecture that looks like Figure 6–2. In this diagram, WebSpeed uses requests to each AppServer to return the number of items at each warehouse. If each request takes 3 seconds, then using synchronous requests will take a total of 5 times 3 seconds, resulting in approximately 15 seconds total for the request, allowing for some network overhead. If you used asynchronous requests, then the total time would be slightly longer than 3 seconds and nowhere near 15 seconds.

For more details on how to write asynchronous AppServer programs, see *OpenEdge Application Server: Developing AppServer Applications*.

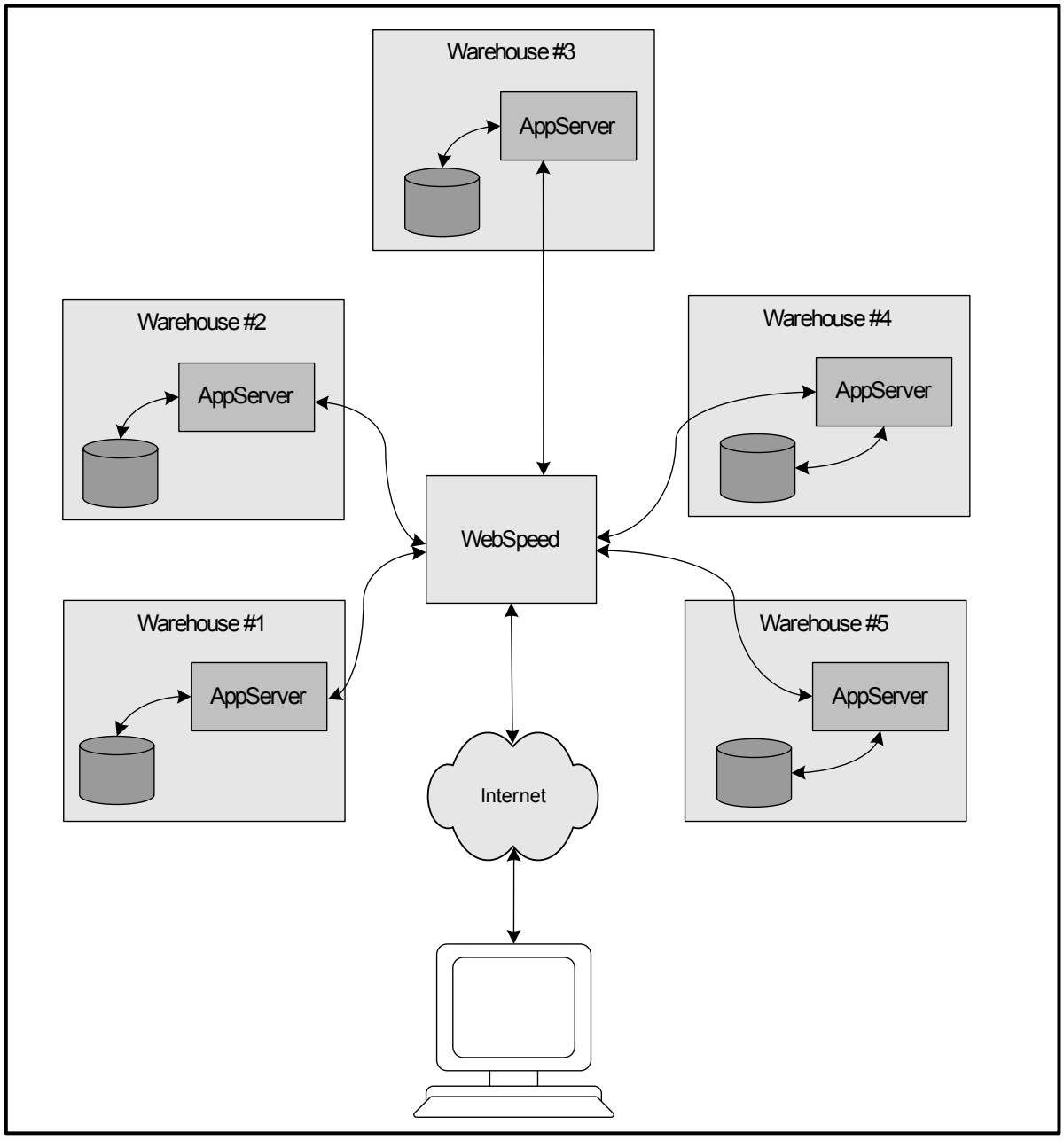


Figure 6–2: Asynchronous requests

AppServer configuration procedures

AppServer configuration procedures are Progress 4GL procedures that you can specify when you configure an AppServer and that run at specific times during an AppServer's lifetime. There are three types of procedures:

- Startup and shutdown procedures
- Connect and disconnect procedures
- Activate and deactivate procedures

Each of these procedures is run at specific times during the lifetime of an AppServer process with the idea of simplifying the process of deploying an application with the AppServer.

Because these procedures are standard 4GL programs, they can do anything the developer wants to do. But, this can cause performance issues if the developer adds too much code, or the code takes a long time to run, because some of these procedures are executed regularly.

All of these procedures are optional and, depending on the AppServer operating mode, some cannot be used. Table 6–1 shows which procedures you can use for each AppServer operating mode.

Table 6–1: AppServer configuration procedures and operating modes

AppServer operating mode	Startup/ shutdown	Connect/ disconnect	Activate/ deactivate
state-reset	No	Yes	No
state-aware	Yes	Yes	No
stateless	Yes	Yes	Yes

For detailed information on these procedures and how to implement them, see *OpenEdge Application Server: Developing AppServer Applications*.

Startup and shutdown procedures

Startup and shutdown procedures encapsulate logic that executes during the startup and shutdown of an AppServer. This occurs when the AppServer broker launches or kills an server. Functionality that might go in a startup procedure includes connecting to databases, loading the contents of temporary tables to provide a cache, and instantiating certain local persistent procedures and super procedures. In a shutdown procedure, the functionality might include disconnecting databases, saving the contents of temp tables, deleting persistent procedures, and general tidying up of the environment.

The startup procedure is run persistently, so it will stay in the AppServer's memory and can be accessed by the client and AppServer procedures. On the other hand, the shutdown procedure is run nonpersistently, just before the server is killed.

Connect and disconnect procedures

`CONNECT` and `DISCONNECT` procedures encapsulate logic that executes when a client application establishes and terminates a connection with an AppServer. The `CONNECT` procedure is usually used to authenticate the AppServer client user. See the "Controlling AppServer entry points" section on page 4–3 for an example of a `CONNECT` procedure that validates a user. User-specific databases and temp tables can also be created and deleted in these procedures.

The `CONNECT` procedure is run persistently for state-reset and state-aware modes and nonpersistently for stateless. The `DISCONNECT` procedure is always run nonpersistently.

Activate and deactivate procedures

When a client application sends remote procedure requests to a stateless AppServer, each request might execute on a different server. To maintain application continuity between requests, you might need to establish some application-specific resources or context before each request and discard the resources and context after each request. The Activate and Deactivate procedures help to manage these resources and context more easily.

Application performance tuning

It is good practice to make sure that the database is performing adequately, and then move your tuning focus to the application. You might get a 15-20% benefit from a database server tweak, but you can gain a 100% benefit from a code or database schema change.

Trying to improve performance of a WebSpeed application mainly deals with making the application scale well. The user at the other end of a dial-up modem will usually not notice small performance issues as the Internet itself is probably the slowest link in the entire process. But, allowing the agent to quickly finish its request and be ready for the next allows for better scalability. If a WebSpeed program takes five seconds to run, it is quite fast, and the customer will be happy. If it could finish in 2.5 seconds, then the customer will still be happy, and the WebSpeed server could cope with twice as many requests per minute than before. This is what you should be aiming for: high performance WebSpeed applications that allow good scalability.

AppServer applications tend to be more interactive so scalability is not normally the highest goal—outright speed is. Making the application faster is usually a combination of understanding the database structures and writing efficient code to access these structures. Plus, you need a good understanding of how the AppServer works, so calls to the AppServer are kept to a minimum.

Two very simple things can make a difference to the performance of a WebSpeed or AppServer application. The first is to use shared r-code libraries and have these on a local disk; and the second is to move the WebSpeed agent or AppServer temporary files onto a separate disk for other high access files, like the UNIX swap area, or the database's BI files. R-code libraries are documented in *OpenEdge Deployment: Managing 4GL Applications*. Making these "shared" will minimize the amount of machine memory used by WebSpeed agents and/or AppServers. This is because shared r-code libraries are loaded into memory once and all the clients (WebSpeed, AppServer, batch, and host-based) will share the one copy. Standard r-code libraries are loaded by each client that accesses them, so a 1MB r-code library used by 25 WebSpeed agents and 100 AppServers will save around 124MB of main memory. Use the `-makeshared` option on the `PROLIB` command to turn a standard r-code library into a shared r-code library.

Fathom Management has facilities to measure and report on the performance of WebSpeed and AppServer application code. The **WebSpeed/AppServer Application Profile** report shows the application code that has run and the maximum and average execution times. This is a good start in finding under-performing application code.

You should review the application code to determine if it is using standard practice for writing good code:

- Does the code use NO-UNDO variables wherever possible?
- Does the code use ASSIGN commands to group micro-transactions?
- Does the code keep transactions and record scopes as small as possible?
- Does the code use DO, instead of REPEAT, unless transactions are needed?
- Does the code use a variable, instead of calling a function, over and over again? An example follows:

```
DEFINE VARIABLE vParam AS CHAR NO-UNDO.
DEFINE VARIABLE vLoop  AS INT  NO-UNDO.

vParam = INT (GET-FIELD ("Param")).

DO vLoop = 1 TO vPARAM:
  /* CODE */
END.
```

This list is not all-encompassing, but it is a good start. The last item is probably where most performance improvements will be gained.

Also check that all the queries, FOR EACH and FIND use indexes. Before putting code into production, compile the code using the XREF option. Then, examine the XREF file for correct index usage. *OpenEdge Data Management: Database Design* describes how to read the XREF output, and *OpenEdge Development: Progress 4GL Reference* details the XREF option on the COMPILE statement.

Progress 4GL Profiler

A much underused utility that Progress provides is the Profiler. This utility will tell you:

- How long each line of code took to run.
- How many times each procedure or function has been called.
- What percentage of the total time was spent in each procedure.

The Profiler will work with any Progress 4GL client that is Version 8.2A or later. The Profiler tool is supplied as source code and is in the *OpenEdge-install*/src/samples/profiler directory. There is a Word document explaining how to set up and run the utility.

For example, at a customer's site, the Profiler was used to determine why a WebSpeed application would not scale. After running the application for a period of time and investigating the Profiler output, it was found that the URL-ENCODE function was being called numerous times and was consuming around 30% of the total request time. After removing calls to the URL-ENCODE function, where they were not needed, and speeding up the function as well, it ended up consuming less than 5% of the total request time and the WebSpeed application scaled.

When you use the Profiler with WebSpeed or AppServer, make sure that only one AppServer server or WebSpeed agent is running. The output log file for the Profiler is sent to a hard-wired filename. If multiple clients try to write to this file, it will cause problems. To force the number of agents/servers to be 1, set the **Initial**, **Minimum**, and **Maximum server settings** equal to 1 on the **Agent/Server Pool Range** tab in the Progress Explorer.

Web server performance

To make the Web server perform faster, make sure that the machine it is running on is not overloaded and that the Web server software is configured appropriately.

Using HTTP compression

Many Web pages today are at least 25KB in size, and some are as large as 100KB. And this is just the HTML! When you add in the images, most will be in the range of 100 to 150KB. Sending this information down to a Web browser over a dial-up modem will be very slow, as most modems will do around 40KB per second. This means that 150KB will take, at best, 35 seconds to download.

There is a solution to this issue and it does not require any changes to the application or to WebSpeed. The solution is called HTTP compression and it is available in most modern Web browsers and Web servers. It is described in an article at this URL: http://www.webreference.com/internet/software/servers/http/compression/index.html. Check your Web server documentation on how to enable HTTP compression and it should increase performance without impacting too much on the Web server.

Alleviating CPU load

Using HTTP/S to encrypt access to the Web site might cause the CPU in the Web server to become overloaded. This is because the encryption requires a lot of mathematics. To alleviate the CPU load, you can use hardware SSL accelerators.

7

Where to Use Fathom Management when Deploying

This chapter highlights the areas of Fathom Management that should be used during application deployment. It includes the following topics:

- Monitoring AppServer and WebSpeed
- NameServer debugging
- Memory, CPU, and disk monitoring
- Log file management
- Using My Fathom

For complete instructions on using Fathom Management features, see the Fathom Management documentation set.

Monitoring AppServer and WebSpeed

Several Fathom Management features can be used to monitor AppServer and WebSpeed, including:

- Broker and Server Performance views
- Trending and reporting data
- Setting rules and configuring alerts
- HTTP monitor

Broker and Server Performance views

The following Fathom Management views can be used to monitor AppServer and WebSpeed server and broker performance:

- **Broker Performance View** — Shows overall broker information and provides a good general view of how the AppServer/WebSpeed service is running. You should use the **Broker Performance View** to get a rough overview of throughput (**Broker Requests Completed** and the **Average Busy Time**). If there are any requests that have been queued, then this is a hint that you might need to allocate more servers to the pool or increase the speed of the application.
- **Server Performance View** — Shows the total number of servers/agents, the number free, and the number of locked or busy servers/agents, along with the CPU and memory consumption of the servers/agents. The **Server Performance View** can be used to see if the servers are leaking memory and steadily growing in size or if they are consuming too much CPU time.

Figure 7–1 shows WebSpeed **Broker** and **Agent** (server) views. The AppServer views are identical in content.

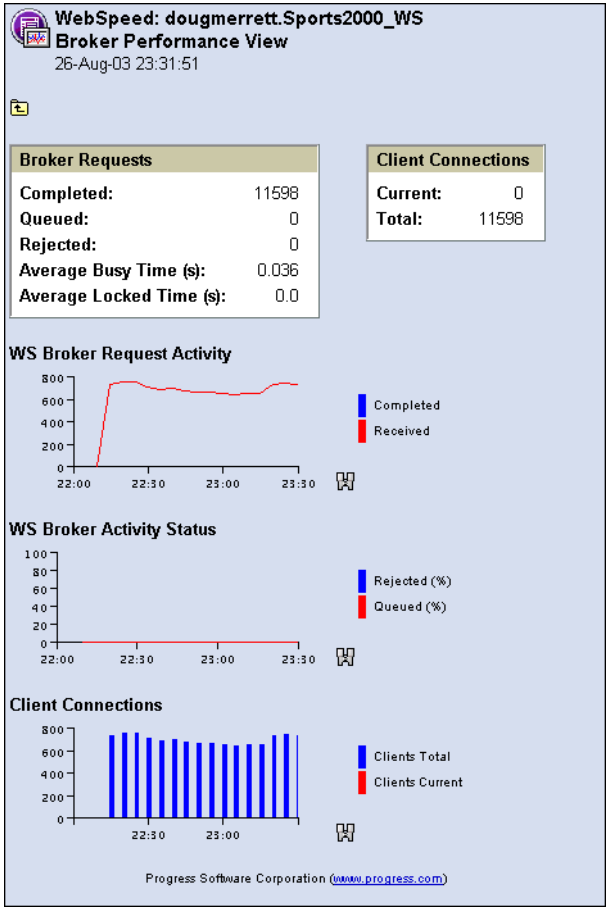


Figure 7–1: Broker Performance View

Trending and reporting data

All of this information can be saved in the FathomTrendDatabase, and reports can be run to show historical comparisons to see if there is a general slowdown, or if at certain times of the week/month, CPU usage is extreme. Make sure you have checked the **Trend Performance Data** check box in the monitoring plan to capture this data, as shown in Figure 7–2. Although this example shows a WebSpeed server, the AppServer setting is configured in exactly the same way.

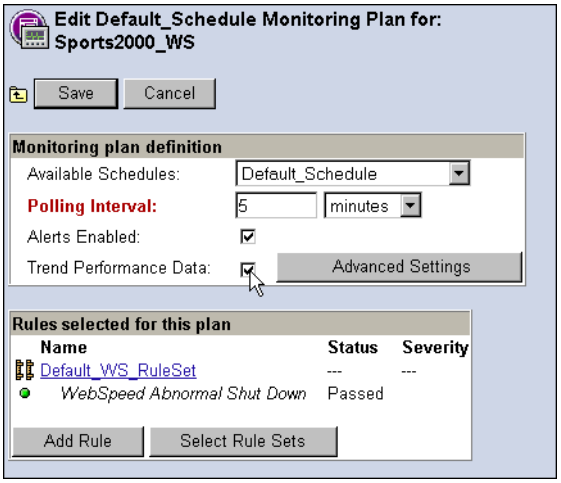


Figure 7–2: Trending performance data

The **Application Profile Report** is a very powerful report that shows the number of times programs are run and the average and maximum time taken for each program. The names of the programs can be configured by using the “matches,” “begins with,” or “literal” values. Use “matches *” to get all programs. Since the FathomTrendDatabase is an OpenEdge database, you can also write your own reports if the standard reports do not suit your needs. The source for the standard reports is included in the `FATHOM/src/fathom4gl-p.pl` file.

You will need to copy this file to a temporary directory and then extract the `.P` files using the `PROLIB` utility documented in *OpenEdge Deployment: Managing 4GL Applications*. Use these programs as an example, or just modify them. See the *Reporting Guide* for more information on customizing Fathom reports.

Setting rules and configuring alerts

After gathering this information for a number of weeks or months, you will have built up a knowledge base from which you can then determine the standard for average run times, memory utilization, and CPU utilization.

This information can then be used to set rules that will alert you to excessive procedure execution time, CPU utilization, queue length, or other key performance indicators.

For the important procedures in your application, configure the **Average Procedure Duration High** rule to alert you to excessive procedure execution times for the WebSpeed or AppServer procedure. By default, this rule is not configured because you need to set it for individual procedures. You can add the rule to the `Default_AS_RuleSet` and the `Default_WS_RuleSet`, or create your own rule set.

An example of setting this rule is shown in Figure 7–3. It shows that the **Default_Mail_Action** action will be thrown after one poll where the average duration for `DoSearch.p` exceeds 75 milliseconds.

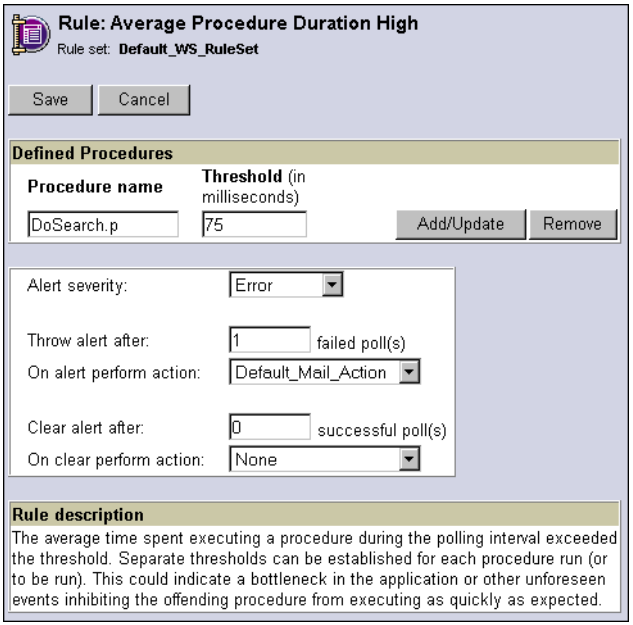


Figure 7–3: Average Procedure Duration High rule

The following rules might also be useful in a deployment environment to help you identify a possible runaway process or a possible memory leak: **Process CPU High**, **Process Resident Memory High**, and **Process Virtual Memory High**.

The **Queued Request Percent High** rule is very useful for checking that the AppServer or WebSpeed servers are not being overloaded with requests. The **Queued Request Percentage** should be kept as low as possible. If the queue is caused by the agents or servers taking too long to return a response, then you need to determine why—is it because the application is running slowly (average procedure duration), or is it due to a machine overload (CPU/Memory usage)? To reduce the queue length, you will need to increase the number of AppServers or WebSpeed agents that the broker can launch and/or increase the speed of the application code by adding more memory, CPU, or modifying the code or database structure to make the code run faster.

Using the Configuration Advisor

You can calculate the values used to set the comparisons for all the above rules by yourself or you can use the **Configuration Advisor** tool. This tool uses data collected over a period of time and recommends settings for the selected rules. You should use the Advisor's settings until you get a better feel for the way the application is running.

HTTP monitor

The Fathom resource monitor for HTTP lets you monitor the state of the Web server and WebSpeed application. You can configure the monitor to check if the Web server is alive and also check if it is returning correct information by using the **Content rule definition** section of the monitoring plan.

For information on how to configure these Fathom features, see the *Resource Monitoring Guide* and the *OpenEdge Server Management Guide*.

NameServer debugging

There are a number of rules that will aid in the debugging of OpenEdge server-based applications when using the NameServer. These are the **Duplicate Broker UUID**, **Broker Timeout**, and the **AppService Not Found** rules.

The **Duplicate Broker UUID** rule is very useful if you need to edit the `ubroker.properties` file manually, because it will alert you if you've forgotten to use `genuuid` to create the UUID when creating a new broker.

Broker Timeout alerts occur when the broker does not send the keep-alive message within the timeout period. This communications process is covered in Chapter 5, "Configuration of the NameServer." This alert usually means that the broker has died and has not tidied up after itself.

The **AppService Not Found** alert occurs when clients are requesting services that are not provided by any brokers that are registered with the NameServer. This can be caused by a misconfiguration of the client application in the case of AppServer requests and from malformed URLs when using WebSpeed. If you are hosting a public Web site using WebSpeed and you are receiving these alerts, it might be because hackers are probing to find which WebSpeed servers are enabled.

Memory, CPU, and disk monitoring

The memory, CPU and disk monitoring feature is useful for tracking usage trends over time. If the virtual memory used is constantly high, adding more real memory to the machine would help with performance as swapping is reduced. Likewise, if the CPU is constantly running at 100%, then adding more CPUs (or better coding) can aid in increasing performance. In general, more CPUs are better than faster CPUs. The reason is that with more CPUs you get better parallelism. Disk storage prices have tumbled and the size of disks has increased dramatically. Having many disks and a large amount free space on these disks will improve performance. Putting high-access parts of your system onto separate disks will increase throughput. Adding hardware RAID 0/1 (mirroring and striping) will also help. Configuring the monitoring of these resources is covered in the *Resource Monitoring Guide*. See *OpenEdge Revealed: Mastering the OpenEdge Database with Fathom Management* for a detailed discussion of RAID.

Log file management

Log file management is a feature that should be used. The ability for Fathom to keep a check on the log file size and look for errors within the log files is very important when running a production system. It allows the administrator to concentrate on doing real work, rather than scanning files for errors. Log file configuration, management, and alerting are covered in the *Resource Monitoring Guide*.

Using My Fathom

The **My Fathom** feature can be used to show the WebSpeed and AppServer throughput graphs, CPU, and database graphs that allow the administrator to see, in a single glance, if the throughput of any application is starting to fall.

This is done by setting the customized view to show the appropriate graphs from the viewlets available. Figure 7–4 show an example **My Fathom** page, which was configured to detail memory usage. For detailed information about the **My Fathom** page and collections, see the *Resource Monitoring Guide*.

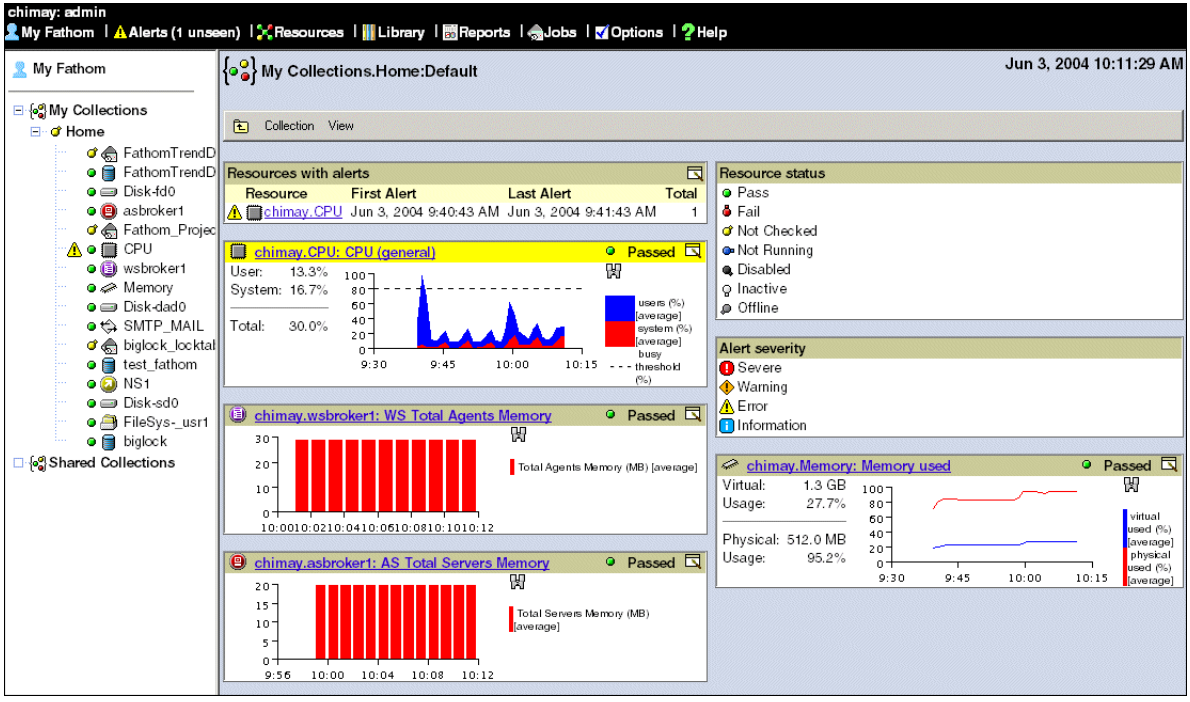


Figure 7–4: My Fathom page

Index

Numbers

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

W