



BEA WebLogic Enterprise

Scaling, Distributing, and Tuning Applications

WebLogic Enterprise 5.1
Document Edition 5.1
May 2000

Copyright

Copyright © 2000 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems, Inc. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems, Inc. DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA Builder, BEA Jolt, BEA Manager, BEA MessageQ, BEA Tuxedo, BEA TOP END, BEA WebLogic, and ObjectBroker are registered trademarks of BEA Systems, Inc. BEA eLink, BEA eSolutions, BEA TAP, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Express, BEA WebLogic Personalization Server, BEA WebLogic Server, Java Enterprise Tuxedo, and WebLogic Enterprise Connectivity are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

Scaling, Distributing, and Tuning Applications

Document Edition	Date	Software Version
5.1	May 2000	BEA WebLogic Enterprise 5.1

Contents

About This Document

What You Need to Know	x
e-docs Web Site	x
How to Print the Document.....	x
Related Information.....	xi
Contact Us!	xi
Documentation Conventions	xii

1. Scaling WebLogic Enterprise Applications

About Scaling WebLogic Enterprise Applications	1-2
Application Scalability Requirements.....	1-2
WebLogic Enterprise Scalability Features	1-2
Scalability Support for WebLogic Enterprise Applications.....	1-3
Using Object State Management	1-4
Object State Models	1-4
Implementing Stateless and Stateful Objects.....	1-6
Replicating Server Processes and Server Groups.....	1-9
About Replicating Server Processes and Server Groups	1-10
Configuration Options.....	1-11
Replicating Server Processes	1-11
Replicating Server Groups	1-12
Using Multithreaded Java Servers (Java only).....	1-13
About Multithreaded Java Servers	1-13
When to Use Multithreaded Java Servers	1-14
Coding Recommendations	1-15
Configuring a Multithreaded Java Server	1-15
Using Factory-based Routing (CORBA only)	1-16

About Factory-based Routing.....	1-16
Characteristics of Factory-based Routing	1-17
How Factory-based Routing Works	1-18
Configuring Factory-based Routing in the UBBCONFIG File.....	1-19
Multiplexing Incoming Client Connections	1-19
IIOP Listener and Handler.....	1-20
Increasing the Number of ISH Processes	1-20

2. Scaling CORBA C++ Server Applications

About Scaling the Production Sample Application.....	2-2
Design Goals	2-2
How the Application Has Been Scaled	2-2
Changing the OMG IDL	2-4
Using a Stateless Object Model.....	2-4
Scaling by Replicating Server Processes and Server Groups.....	2-5
Replicating Server Processes in the Production Application	2-6
Replicating Server Groups in the Production Application	2-8
Configuring Replicated Server Processes and Groups in the Production Application	2-9
Scaling with Factory-based Routing	2-11
About Factory-based Routing in the Production Application	2-11
Configuring Factory-based Routing in the UBBCONFIG File.....	2-12
Implementing Factory-based Routing in a Factory	2-15
What Happens at Run Time	2-16
Additional Design Considerations	2-17
About the Additional Design Considerations.....	2-17
Instantiating the Registrar and Teller Objects	2-18
Ensuring That Student Registration Occurs in the Correct Server Group	2-19
Ensuring That the Teller Object Is Instantiated in the Correct Server Group. 2-21	
Scaling the Application Further.....	2-22

3. Scaling CORBA Java Server Applications

About Scaling the JDBC Bankapp Sample Application	3-2
Design Goals	3-2
How the Application Has Been Scaled	3-2

Scaling with Object State Management	3-3
Scaling by Replicating Server Processes and Server Groups.....	3-4
Replicating Server Processes in the Bankapp Application	3-4
Replicating Server Groups in the Bankapp Application	3-6
Configuring Replicated Server Processes and Groups in the Bankapp Application	3-7
Scaling with Factory-based Routing	3-10
About Factory-based Routing in the Bankapp Application	3-10
Configuring Factory-based Routing in the UBBCONFIG File	3-11
Implementing Factory-based Routing in a Factory	3-12
What Happens at Run Time	3-14
Additional Design Considerations.....	3-14
About the Additional Design Considerations.....	3-15
Instantiating the Teller Object.....	3-15
Ensuring That Account Updates Occur in the Correct Server Group	3-16
Scaling the Application Further.....	3-16

4. Scaling EJB Applications

Scaling Tasks for EJB Providers	4-2
Using Stateless Session Beans	4-2
Minimizing State Information in Stateful Session Beans	4-3
Using Pooled Connections	4-3
Implementing Methods for Bean Persistence.....	4-3
Completing Transactions Efficiently	4-4
Implementing the Process-entity Design Pattern	4-5
Scaling Tasks for Application Assemblers and Deployers	4-5
Organizing EJBs in Groups.....	4-6
Configuring the Persistent Storage Location	4-6
Specifying the Method to Invoke Before Passivation	4-7
Deploying Stateful Session Beans on the IIOP Listener Node	4-7
Configuring the EJB Cache.....	4-7
Scaling Tasks for System Administrators	4-8
Removing Orphan Files for File-based Persistence	4-8
Scaling and Tuning the EJB Cache	4-9

5. Distributing Applications

Why Distribute an Application?	5-2
About Distributing an Application	5-2
Benefits of a Distributed Application	5-2
Characteristics of Distributing an Application	5-3
Using Data-dependent Routing (BEA Tuxedo Servers Only)	5-4
About Data-dependent Routing	5-4
Characteristics of Data-dependent Routing	5-5
Sample Distributed Application	5-5
Configuring the UBBCONFIG File	5-6
About the UBBCONFIG File in Distributed Applications	5-6
Modifying the GROUPS Section	5-7
Modifying the SERVICES Section	5-8
Creating the ROUTING Section	5-10
Example of UBBCONFIG Sections in a Distributed Application	5-10
Configuring the factory_finder.ini (CORBA Applications Only)	5-11
Modifying the Domain Gateway Configuration File to Support Routing	5-11
About the Domain Gateway Configuration File	5-12
Parameters in the DM_ROUTING Section of the DMCONFIG File (BEA Tuxedo Only)	5-12

6. Tuning Applications

Maximizing Application Resources	6-2
When to Use MSSQ Sets (BEA Tuxedo Servers Only)	6-2
Enabling Load Balancing	6-4
About Load Balancing	6-4
Two Ways to Measure Service Performance Time (BEA Tuxedo Servers Only)	6-5
Configuring Replicated Server Processes and Groups	6-5
Configuring Multithreaded Java Servers	6-7
Setting the OPENINFO Parameter	6-7
Configuring the Number of Threads	6-7
Configuring the Number of Concurrent Accessors	6-8
Assigning Priorities to Interfaces or Services	6-9
About Priorities to Interfaces or Services	6-9

Characteristics of the PRIO Parameter	6-9
Bundling Services into Servers (BEA Tuxedo Servers Only)	6-10
About Bundling Services	6-10
When to Bundle Services	6-10
Enhancing Efficiency with Application Parameters.....	6-11
Setting the MAXACCESSERS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES Parameters	6-12
Setting the MAXGTT, MAXBUFTYPE, and MAXBUFSTYPE Parameters 6-13	
Setting the SANITYSCAN, BLOCKTIME, BBLQUERY, and DBBLWAIT Parameters	6-13
Setting Application Parameters	6-14
Determining IPC Requirements	6-15
Measuring System Traffic	6-16
About System Traffic and Bottlenecks	6-17
Example of Detecting a System Bottleneck	6-17
Detecting Bottlenecks on UNIX	6-18
Detecting Bottlenecks on Windows NT	6-19



About This Document

This document explains how to tune and scale CORBA, EJB, and RMI applications that run in the BEA WebLogic Enterprise™ (WLE) environment.

This document covers the following topics:

- Chapter 1, “Scaling WebLogic Enterprise Applications,” describes how to scale CORBA, EJB, and RMI applications that run in the WebLogic Enterprise environment.
- Chapter 2, “Scaling CORBA C++ Server Applications,” describes how to scale CORBA C++ server applications using the Production sample application as an example.
- Chapter 3, “Scaling CORBA Java Server Applications,” describes how to scale CORBA Java server applications using the sample Bankapp application as an example.
- Chapter 4, “Scaling EJB Applications,” describes how to scale WebLogic Enterprise EJB applications.
- Chapter 5, “Distributing Applications,” describes how to distribute applications using the Production and Bankapp sample applications as examples.
- Chapter 6, “Tuning Applications,” describes how to tune applications to optimize performance.

What You Need to Know

This document is intended primarily for application developers who are interested in building scalable C++ and Java applications that run in the WebLogic Enterprise environment. It assumes a familiarity with the WebLogic Enterprise platform and C++ or Java programming.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA Systems, Inc. corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at <http://e-docs.bea.com>.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Enterprise documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at <http://www.adobe.com/>.

Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA Tuxedo®, distributed object computing, transaction processing, C++ programming, and Java programming, see the WebLogic Enterprise *Bibliography* in the WebLogic Enterprise online documentation.

Contact Us!

Your feedback on the BEA WebLogic Enterprise documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Enterprise documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.1 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSUPPORT at www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
<i>italics</i>	Indicates emphasis or book titles.
monospace text	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard. <i>Examples:</i> #include <iostream.h> void main () the pointer psz chmod u+w * \\tux\data\ap .doc tux.doc BITMAP float
monospace boldface text	Identifies significant words in code. <i>Example:</i> void commit ()
<i>monospace italic text</i>	Identifies variables in code. <i>Example:</i> String <i>expr</i>
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. <i>Examples:</i> LPT1 SIGNON OR

Convention	Item
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[]	Indicates optional items in a syntax line. The brackets themselves should never be typed. <i>Example:</i> buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...
	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
...	Indicates one of the following in a command line: <ul style="list-style-type: none"> ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. <i>Example:</i> buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.



1 Scaling WebLogic Enterprise Applications

This topic introduces key concepts and tasks for scaling WebLogic Enterprise applications. This topic includes the following sections:

- About Scaling WebLogic Enterprise Applications
- Using Object State Management
- Replicating Server Processes and Server Groups
- Using Multithreaded Java Servers (Java only)
- Using Factory-based Routing (CORBA only)
- Multiplexing Incoming Client Connections

For more detailed information and examples for different types of WebLogic Enterprise applications, see the following topics:

- Chapter 2, “Scaling CORBA C++ Server Applications”
- Chapter 3, “Scaling CORBA Java Server Applications”
- Chapter 4, “Scaling EJB Applications”
- For RMI applications, see *Using RMI in a WebLogic Enterprise Environment*

About Scaling WebLogic Enterprise Applications

This topic includes the following sections:

- Application Scalability Requirements
- WebLogic Enterprise Scalability Features
- Scalability Support for WebLogic Enterprise Applications

Application Scalability Requirements

Many applications perform adequately in an environment where between 1 to 10 server processes and 10 to 100 client applications are running. However, in an enterprise environment, applications may need to support hundreds of execution contexts (where the context can be a thread or a process), tens of thousands of client applications, and millions of objects at satisfactory performance levels.

Subjecting an application to exponentially increasing demands quickly reveals any resource shortcomings and performance bottlenecks in the application. Scalability is therefore an essential characteristic of WebLogic Enterprise applications.

You can build highly scalable WebLogic Enterprise applications by:

- Adding parallel processing capability to enable the WebLogic Enterprise domain to process multiple client requests simultaneously.
- Sharing the processing load on the server applications across multiple machines.

WebLogic Enterprise Scalability Features

WebLogic Enterprise supports large-scale application deployments by:

- Optimizing object state management

- Load balancing objects and requests across replicated server processes and server groups
- For Java applications, using multithreaded Java servers, which are appropriate for certain types of applications and processing environments
- For CORBA applications, using factory-based routing
- Using data-dependent routing (Tuxedo only)
- Multiplexing incoming client connections

Scalability Support for WebLogic Enterprise Applications

Table 1-1 shows how WebLogic Enterprise scalability features support each type of WebLogic Enterprise application.

Table 1-1 Supported Scalability Features for WebLogic Enterprise Applications

WebLogic Enterprise Feature	CORBA C++	CORBA Java	EJB	RMI
Object state management	Supported	Supported	Supported	Not supported
Replicating server processes and server groups	Supported	Supported	Supported	Supported
Using multithreaded servers	Not supported	Supported	Supported	Supported
Factory-based routing	Supported	Supported	Not supported	Not supported
Multiplexing incoming client connections	Supported	Supported	Supported	Supported

Notes: CORBA and EJB applications require slightly different configuration parameters in the `UBBCONFIG` file. For more information, see “Creating a Configuration File” in the *Administration Guide*.

For RMI applications, callback objects are not scalable because they are not subject to WebLogic Enterprise administration. For more information about callback objects, see “Using RMI with Client-side Callbacks” in *Using RMI in a WebLogic Enterprise Environment*.

Using Object State Management

This topic includes the following sections:

- CORBA Object State Models
- EJB Object State Models
- Implementing Stateless and Stateful Objects

Object state management is a fundamental concern of large-scale client/server systems because it is critical that such systems achieve optimized throughput and response time. For more detailed information about using object state management, see the following topics:

- For CORBA C++ applications, see “Using a Stateless Object Model” on page 2-4.
- For CORBA Java applications, see “Scaling with Object State Management” on page 3-3.
- For EJB applications, see “Scaling Tasks for EJB Providers” on page 4-2.
- For all WebLogic Enterprise applications, see the technical article *Process-Entity Design Pattern*.

Object State Models

This topic describes the following object state models:

- CORBA Object State Models
- EJB Object State Models
- RMI Object State Models

CORBA Object State Models

WebLogic Enterprise CORBA supports three object state management models:

- Method-bound Objects
- Process-bound Objects
- Transaction-bound Objects

For more information about these models, see “Server Application Concepts” in *Creating CORBA C++ Server Applications*.

Method-bound Objects

Method-bound objects are loaded into the machine’s memory only for the duration of the client invocation. When the invocation is complete, the object is deactivated and any state data for that object is flushed from memory. In this document, a method-bound object is considered to be a *stateless object*.

You can use method-bound objects to create a stateless server model in your application. By using a stateless server model, you move requests that are already directed to active objects to any available server, which allows concurrent execution for thousands and even millions of objects. From the client application view, all the objects are available to service requests. However, because the server application maps objects into memory only for the duration of client invocations, few of the objects managed by the server application are in memory at any given moment.

Process-bound Objects

Process-bound objects remain in memory beginning when they are first invoked until the server process in which they are running is shut down. A process-bound object can be activated upon a client invocation or explicitly before any client invocation (a *preactivated* object). Applications can control the deactivation of process-bound objects. In this document, a process-bound object is considered to be a *stateful object*.

When appropriate, process-bound objects with a large amount of state data can remain in memory to service multiple client invocations, thereby avoiding reading and writing the object’s state data on each client invocation.

Transaction-bound Objects

Transaction-bound objects can also be considered stateful because, within the scope of a transaction, they can remain in memory between invocations. If the object is activated within the scope of a transaction, the object remains active until the

transaction is either committed or rolled back. If the object is activated outside the scope of a transaction, its behavior is the same as that of a method-bound object (it is loaded for the duration of the client invocation).

EJB Object State Models

WebLogic Enterprise implements the Enterprise JavaBeans 1.1 Specification published by Sun Microsystems, Inc. WebLogic Enterprise fully supports the three EJB types defined in the specification:

- **Stateless session beans** are stateless objects and are analogous to *method-bound* objects in CORBA applications.
- **Stateful session beans** are stateful objects and are analogous to *process-bound* objects in CORBA applications.
- **Entity beans** are stateful objects and are analogous to *process-bound* objects in CORBA applications.

For more information about these EJB types, see “Types of Beans Supported in WebLogic Enterprise” in “The WebLogic Enterprise JavaBeans Programming Environment” topic in *Getting Started*. For more information about object state management in EJB applications, see “Scaling Tasks for EJB Providers” on page 4-2.

RMI Object State Models

In RMI applications, a conversational state exists between the client application and the object instance. RMI objects remain in memory beginning when they are first created for as long as the object exists or until the server process in which they are running is shut down. For more information about RMI applications, see *Using RMI in a WebLogic Enterprise Environment*.

Implementing Stateless and Stateful Objects

In general, application developers need to balance the costs of implementing stateless objects against the costs of implementing stateful objects.

About Stateless and Stateful Objects

The decision to use stateless or stateful objects depends on various factors. In the case where the cost to initialize an object with its durable state is expensive—because, for example, the object’s data takes up a great deal of space, or the durable state is located on a disk very remote from the servant that activates it—it may make sense to keep the object stateful, even if the object is idle during a conversation. In the case where the cost to keep an object active is expensive in terms of machine resource usage, it may make sense to make such an object stateless.

By managing object state in a way that is efficient and appropriate for your application, you can maximize your application’s ability to support large numbers of simultaneous client applications that use large numbers of objects. The way that you manage object state depends on the specific characteristics and requirements of your application:

- For CORBA applications, you do this by assigning the `method activation` policy to these objects, which has the effect of deactivating idle object instances so that machine resources can be allocated to other object instances.
- For EJB applications, you use stateless session beans when possible, because they are load balanced, on a per-request basis, within a group and across groups.

When to Use Stateless Objects

Stateless objects generally provide good performance and optimal usage of server resources, because server resources are never used when objects are idle. Using stateless objects is a good approach to implementing server applications and are particularly appropriate when:

- The client application waits for user input between invocations on the object.
- The client request contains all the data needed by the server application, and the server can process the client request using only that data.
- The object has high access rates, but low access rates from any one particular client application.

By making an object stateless, you can generally assure that server application resources are not being reserved unnecessarily while waiting for input from the client application.

An application that employs a stateless object model has the following characteristics:

- Information about and associated with an invocation is not maintained after the server application has finished executing a client request.
- An incoming client request is sent to the first available server process. After the request has been satisfied, the application state disappears and the server application is available for another client application request.
- Durable state information for the object exists outside the server process. With each invocation on this object, the durable state is read into memory.
- Successive requests on an object from a given client application may be processed by a different server process.
- The overall system performance of a machine that is running stateless objects is usually enhanced.

When to Use Stateful Objects

A stateful object, once activated, remains in memory until a specific event occurs, such as the process in which the object exists is shut down, or the transaction in which the object is activated is completed.

Using stateful objects is recommended when:

- An object is used frequently by a large number of client applications, such as long-lived, well-known objects. When the server application keeps these objects active, the client application typically experiences minimal response time in accessing them. These active objects are shared by many client applications, and therefore relatively few objects of this type exist in memory.

Note: You should carefully consider how objects will potentially be involved in a transaction. An object can be bound to a particular process temporarily (transaction-bound) or permanently (process-bound). An object that is involved in a transaction cannot be invoked by another client application or object (WebLogic Enterprise will likely return an error indicating that the object is busy). Stateful objects that are intended to be used by a large number of client applications can create bottlenecks if they are involved in transactions frequently or for long durations.

- A client application must invoke successive operations on an object to complete a transaction, and the client application is not idle while it waits for user input between invocations. If the object were deactivated between invocations, there would be a degradation of response time because state would be written and read

between each invocation. In EJB applications, stateful objects can be passivated at any time. Such behavior may not be appropriate for transactions. You should consider holding server resources in exchange for better response time.

Stateful objects have the following behavior:

- State information is maintained between server invocations, and the object typically remains dedicated to a given client application for a specified duration. Even though data is sent and received between the client and server applications, the server process maintains additional context or application state information in memory.
- When one or more stateful objects use a lot of machine resources, server performance for tasks and processes not associated with the stateful object may be lower than with a stateless server model.

For example, if an object has a lock on a database and is caching large amounts of data in memory, that database and the memory used by that stateful object are unavailable to other objects, potentially for the entire duration of a transaction.

Replicating Server Processes and Server Groups

This topic includes the following sections:

- About Replicating Server Processes and Server Groups
- Configuration Options
- Replicating Server Processes
- Replicating Server Groups

For more detailed information about replicating server processes and server groups, see the following topics:

- “Configuring Replicated Server Processes and Groups” on page 6-5

- For CORBA C++ applications, see “Scaling by Replicating Server Processes and Server Groups” on page 2-5
- For CORBA Java applications, see “Scaling by Replicating Server Processes and Server Groups” on page 3-4
- For EJB applications, see “Scaling Tasks for System Administrators” on page 4-8

About Replicating Server Processes and Server Groups

The WebLogic Enterprise environment allows CORBA objects and EJBs to be deployed across multiple servers to provide additional failover reliability and to split the client’s workload through load balancing. WebLogic Enterprise load balancing is enabled by default. For more information about configuring load balancing, see “Enabling Load Balancing” on page 6-4. For more information about distributing the application workload using BEA Tuxedo features, see Chapter 5, “Distributing Applications.”

The WebLogic Enterprise architecture provides the following server organization:

- **Groups.** Individual servers can be combined to form a group. A group of servers runs on a single machine. Typically, the servers in a group access common resources (such as a database).
- **Domains.** Machines can be combined to form a domain. A domain is administered centrally. Multiple domains are administered separately. Domains can also be interconnected and requests can be transparently routed from one domain to another. However, each domain is independently administered.

This architecture allows new servers, groups, or machines to be dynamically added or removed, to adapt the application to high- or low-demand periods, or to accommodate internal changes required to the application. The WebLogic Enterprise run time provides load balancing and failover by routing requests across available servers.

System administrators can scale a WebLogic Enterprise application by:

- **Replicating Server Processes.** Increase the number of server processes to support more active objects within a group and load balancing among servers.

- **Replicating Server Groups.** Increase the number of server groups so that WebLogic Enterprise can balance the load by distributing processing requests across multiple server machines.

Configuration Options

You can configure server applications as:

- A single machine with one or more server processes implementing one or more interfaces. For Java, the servers can be single-threaded or multithreaded. For C++, the servers are single-threaded only.
- Multiple machines with multiple server processes and multiple interfaces.

You can add more parallel processing capability to client/server applications by replicating server processes or add more threads. You can add more server groups to split processing across resource managers. For CORBA applications, you can implement factory-based routing, as described in “Using Factory-based Routing (CORBA only)” on page 1-16.

Replicating Server Processes

System administrators can scale an EJB application by replicating the servers to support more concurrent active objects, or process more concurrent requests, on the server node. To configure replicated server processes, see “Configuring Replicated Server Processes and Groups” on page 6-5.

Benefits

The benefits of using replicated server processes include:

- Load balancing incoming requests.
- Processing client requests on any server within a group. As requests arrive in the WebLogic Enterprise domain for the server group, WebLogic Enterprise routes the request to the least busy server process within that group.

- Improving the server application's performance by using multiple server processes. Instead of having one server process handling one client request at one time, multiple server processes are available to handle multiple client requests simultaneously.
- Providing failover protection in the event that one of the server processes stops.

Guidelines

To achieve the maximum benefit of using replicated server processes, make sure that the CORBA objects or entity beans instantiated by your server application have unique object IDs. This allows a client invocation on an object to cause the object to be instantiated on demand, within the bounds of the number of server processes that are available, and not queued up for an already active object.

You should also consider the trade-off between providing better application recovery by using multiple processes versus more efficient performance using threads (for some types of application patterns and processing environments).

Better failover occurs only when you add processes, *not* threads. For information about using single-threaded and multithreaded Java servers, see “When to Use Multithreaded Java Servers” on page 1-14.

Replicating Server Groups

Server groups are unique to WebLogic Enterprise and are key to the scalability features of WebLogic Enterprise. A group contains one or more servers on a single node. System administrators can scale a WebLogic Enterprise application by replicating server groups and configuring load balancing within a domain.

Replicating a server group involves defining another server group with the same type of servers and resource managers to provide parallel access to a shared resource (such as a database). CORBA applications, for example, can use factory-based routing to split processing across the database partitions.

The `UBBCONFIG` file specifies how server groups are configured and where they run. By using multiple server groups, WebLogic Enterprise can:

- Spread the processing load for a given application or set of applications across additional machines.

- For CORBA applications, use factory-based routing to send one set of requests on a given interface to one group, and another set of requests on the same interface to another group.

To configure replicated server groups, see “Configuring Replicated Server Processes and Groups” on page 6-5.

Using Multithreaded Java Servers (Java only)

This topic includes the following sections:

- About Multithreaded Java Servers
- When to Use Multithreaded Java Servers
- Coding Recommendations
- Configuring a Multithreaded Java Server

For instructions on how to configure Java servers for multithreading, see “Configuring Multithreaded Java Servers” on page 6-7.

Note: C++ servers are single-threaded only.

About Multithreaded Java Servers

System administrators can scale a WebLogic Enterprise application by enabling multithreading in Java servers, and by tuning configuration parameters (the maximum number of server worker threads that can be created) in the application’s `UBBCONFIG` file.

WebLogic Enterprise Java supports the ability to configure multithreaded WebLogic Enterprise Java applications. A multithreaded WebLogic Enterprise Java server can service multiple object requests simultaneously, while a single-threaded WebLogic Enterprise Java server runs only one request at a time. Running a WebLogic Enterprise

Java server in multithreaded mode or in single-threaded mode is transparent to the application programmer. Programs written to WebLogic Enterprise Java run without modification in both modes.

Server worker threads are started and managed by the WebLogic Enterprise Java software rather than an application program. Internally, WebLogic Enterprise Java manages a pool of available server worker threads. If a Java server is configured to be multithreaded, then when a client request is received, an available server worker thread from the thread pool is scheduled to execute the request. Each active object has an associated thread, and while the object is active, the thread is busy. When the request is complete, the worker thread is returned to the pool of available threads.

Note: In this release, you should *not* establish multiple threads programmatically in your server implementation code. Only worker threads that are created by the run-time WebLogic Enterprise Java server software can access the WebLogic Enterprise Java infrastructure, which means that your Java server application should not create a Java thread from a worker thread and then attempt to begin a new transaction in the thread. You can, however, start threads in your server application to perform other, non-WebLogic Enterprise operations.

When to Use Multithreaded Java Servers

Deploying multithreaded Java servers is appropriate for many, but not all, WebLogic Enterprise Java applications. The potential for a performance gain from a multithreaded Java server depends on whether:

- The application is running on a single- or a multiprocessor machine.
- The application is CPU-intensive or I/O-intensive.

If the application is running on a single-processor machine and the application is CPU-intensive only (for example, without any I/O), in most cases the multithreaded Java server will not increase performance. In fact, due to the overhead of switching between threads, using a multithreaded Java server in this configuration might result in a performance loss rather than a gain.

In general, however, WebLogic Enterprise Java applications almost always perform better when running on multithreaded Java servers. Multiple multithreaded servers should be configured to distribute the load across servers. If only a single server is configured, that server's queue could fill up quickly.

Coding Recommendations

The code used in a multithreaded WebLogic Enterprise server application appears the same as a single-threaded application. However, if you plan to configure your Java server applications to be multithreaded, or you want to have the option do so in the future, consider the following recommendations:

- Do not start threads in your Java server code, and keep threading transparent in your source files.
- Write thread-safe code in your server and client code. Use standard Java synchronization techniques to make sure that static variables are properly synchronized. For more information about Java synchronization techniques, see the Java Language Specification published by Sun Microsystems, Inc.
- Configure the Java server as single-threaded if your application uses JNI code to access ATMI. Alternatively, a multithreaded Java server can access Tuxedo services using Java Enterprise Tuxedo (JET), as described in Using Java Enterprise Tuxedo.
- Configure the Java server as multithreaded if an XA-enabled version of Java server is built using `buildXAJS`. The server *must* be configured to support multithreaded mode.
- Include one of the following identifiers in each message if your client or server application sends messages to the user log (ULOG):
 - Object ID
 - Thread name
 - Transaction ID (if the object is transactional)

Configuring a Multithreaded Java Server

To configure a multithreaded Java server, you change settings in the application's `UBBCONFIG` file. For information about defining the `UBBCONFIG` parameters to implement a multithreaded Java server, see “Configuring Multithreaded Java Servers” on page 6-7.

Using Factory-based Routing (CORBA only)

This topic includes the following sections:

- About Factory-based Routing
- How Factory-based Routing Works
- Configuring Factory-based Routing in the UBBCONFIG File

This topic introduces factory-based routing in WebLogic Enterprise CORBA applications. For more detailed information about using factory-based routing, see the following topics:

- For CORBA C++ applications, see “Configuring Factory-based Routing in the UBBCONFIG File” on page 2-12.
- For CORBA Java applications, see “Configuring Factory-based Routing in the UBBCONFIG File” on page 3-11.

About Factory-based Routing

Factory-based routing is a feature that lets you send a client request to a specific server group. Using factory-based routing, you can distribute that processing load for a given application across multiple machines, because you can determine the group and machine in which a given object is instantiated.

Routing is performed when a factory creates an object reference. The factory specifies field information in its call to the WebLogic Enterprise TP Framework to create an object reference. The TP Framework executes the routing algorithm based on the routing criteria that you define in the `ROUTING` section of an application's `UBBCONFIG` file. The resulting object reference has, as its target, an appropriate server group for the handling of method invocations on the object reference. Any server that implements the interface in that server group is eligible to activate the servant for the object reference.

The activation of CORBA objects can be distributed by server group based on defined criteria, in cooperation with a system designer. Different implementations of CORBA interfaces can be supplied in different groups. This feature enables you to replicate the same CORBA interface across multiple server groups, based on defined, group-specific differences.

The system designer of the application must communicate the factory-based routing criteria to the system administrator. In the BEA Tuxedo system, an FML field used for a service invocation can be used for routing. You can independently discover this information because there is no service request message data or associated buffer information available for routing. Routing is performed at the factory level and not on a method invocation on the target CORBA object.

The primary benefit of factory-based routing is that it provides a simple means to scale up an application, and invocations on a given interface in particular, across a growing deployment environment. Distributing the deployment of an application across additional machines is strictly an administrative function that does not require you to recode or rebuild the application.

Characteristics of Factory-based Routing

Factory-based routing has the following characteristics:

- An implementation of a particular CORBA interface can exist in more than one server process, as shown in “Configuring Factory-based Routing in the UBBCONFIG File” on page 2-12.
- Multiple CORBA interfaces can reside in a single server group.
- All server processes in a particular server group do *not* need to use the same CORBA interfaces.
- The factory object implementation can indirectly control the location of the created CORBA object by supplying application-specific routing information.
- Routing uses the Bulletin Board criteria and occurs in a server call.
- All instances that offer a given interface within a group must support the same version of the implementation.

How Factory-based Routing Works

To implement factory-based routing, you change the way your factories create object references.

- You coordinate with the system designer to determine the fields and values to be used as the basis for routing.
- For each interface, you need to configure factory-based routing. The interface definition for the factory must specify the parameter that represents the routing criteria used to determine the group ID.
- In the `UBBCONFIG` file, you need to define the following information:
 - Routing criteria identifier for a CORBA interface in the `INTERFACES` section.
 - As many server groups as are required for distributing the system in the `GROUPS` section.
 - Routing criteria in the `ROUTING` section.
 - Groups, machines, and databases as required.
- An object with a given interface and OID can be simultaneously active in two different groups *if* those two groups both contain the same object implementation. This can be avoided if your factories generate unique OIDs. To guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, you must either:
 - Use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or
 - Configure your domain so that a given object implementation is in only one group.

If multiple clients have an object reference that contains a given interface name and OID, the reference will always be routed to the same object instance.

Thereafter, the object reference will contain additional information that is used to provide an indication of where the target server exists. Factory-based routing is performed once per CORBA object, when the object reference is created.

Configuring Factory-based Routing in the UBBCONFIG File

Routing criteria specify the data values used to route requests to a particular server group. To configure factory-based routing, you define routing criteria in the `ROUTING` section of the `UBBCONFIG` file (for each interface for which requests are routed). For more detailed information about configuring factory-based routing, see the following topics:

- For CORBA C++, see “Configuring Factory-based Routing in the UBBCONFIG File” on page 2-12.
- For CORBA Java, see “Configuring Factory-based Routing in the UBBCONFIG File” on page 3-11.

To configure factory-based routing across multiple domains, you must also configure the `factory_finder.ini` file to identify factory objects that are used in the current (local) domain but that are resident in a different (remote) domain. For more information, see “Configuring Multiple Domains (WebLogic Enterprise System)” in the *Administration Guide*.

Multiplexing Incoming Client Connections

This topic includes the following sections:

- IIOP Listener and Handler
- Increasing the Number of ISH Processes

System administrators can scale a WebLogic Enterprise application by increasing, in the `UBBCONFIG` file, the number of incoming client connections that an application site supports. WebLogic Enterprise provides a multicontexted, multistated gateway of listener/handlers to handle the multiplexing of all the requests issued by the client.

IIOP Listener and Handler

The IIOP Listener (ISL) enables access to WebLogic Enterprise objects by remote WebLogic Enterprise clients that use IIOP. The ISL is a process that listens for remote clients requesting IIOP connections. The IIOP Handler (ISH) is a multiplexor process that acts as a surrogate on behalf of the remote client. Both the ISL and ISH run on the application site. An application site can have one or more ISL processes and multiple associated ISH processes. Each ISH is associated with a single ISL.

The client connects to the ISL process using a known network address. The ISL balances the load among ISH processes by selecting the best available ISH and passing the connection directly to it. The ISL/ISH manages the context on behalf of the application client. For more information about ISL and ISH, see the description of ISL in the *Commands, Systems Processes, and MIB Reference*.

Increasing the Number of ISH Processes

System administrators can scale a WebLogic Enterprise application by increasing the number of ISH processes on an application site, thereby enabling the ISL to load balance among more ISH processes. By default, an ISH can handle up to 10 client connections. To increase this number, pass the optional `CLOPT -x mpx-factor` parameter to the ISL command, specifying in *mpx-factor* the number of ISH client connections each ISH can handle (up to 4096), and therefore the degree of multiplexing, for the ISH. Increasing the number of ISH processes may affect application performance as the application site services more concurrent processes.

System administrators can tune other ISH options as well to scale WebLogic Enterprise applications. For more information, see the description of ISL in the *Commands, Systems Processes, and MIB Reference*.

2 **Scaling CORBA C++ Server Applications**

This topic includes the following sections:

- About Scaling the Production Sample Application
- Changing the OMG IDL
- Using a Stateless Object Model
- Scaling by Replicating Server Processes and Server Groups
- Scaling with Factory-based Routing
- Additional Design Considerations
- Scaling the Application Further

Using the Production sample application as an example, this topic demonstrates scaling an WebLogic Enterprise CORBA C++ application to increase its processing capability. Before you begin, be sure to read:

- Chapter 1, “Scaling WebLogic Enterprise Applications,” for a comprehensive introduction to tuning and scaling WebLogic Enterprise applications.
- Production Sample Application in the WebLogic Enterprise online documentation.

About Scaling the Production Sample Application

The Production sample application provides the same end-user functionality as the Wrapper sample application. The Production sample application demonstrates how to use features of the WebLogic Enterprise software to scale an existing WebLogic Enterprise application.

This section includes the following topics:

- Design Goals
- How the Application Has Been Scaled

Design Goals

The primary design goal of the Production sample application is to significantly increase the number of client applications it can accommodate by:

- Processing, in parallel and on one machine, client requests on multiple objects that implement the same interface.
- Directing requests on behalf of certain students to one machine, and other students to other machines.
- Adding more machines to share the processing load.

How the Application Has Been Scaled

To accommodate these design goals, the Production sample application has been scaled by:

- Implementing a stateless object model to scale up the number of client requests the server process can manage simultaneously.

- Replicating the University, Billing, and BEA Tuxedo Teller Application server processes within the groups in which they are configured (the `ORA_GRP` and `APP_GRP` server groups defined in the `UBBCONFIG` file).
- Replicating the `ORA_GRP` and `APP_GRP` server groups on an additional server machine, Production Machine 2, and also partitioning the database.
- Assigning unique object IDs (OIDs) to the following objects so that they can be instantiated multiple times simultaneously in their respective groups.
 - `RegistrarFactory`
 - `Registrar`
 - `TellerFactory`
 - `Teller`

This makes these objects available on a per-client application (and not per-process) basis, thereby accommodating a parallel processing capability.

- Implementing factory-based routing to direct client requests on behalf of some students to one machine, and other students to another machine.

Note: To make the Production sample application easy to use, this application is configured on the WebLogic Enterprise software kit to run on one machine, using one database. The examples shown in this chapter, however, show running this application on two machines using two databases.

The Production sample application is designed so that it can be configured to run on several machines and to use multiple databases. Changing the configuration to multiple machines and databases involves modifying the `UBBCONFIG` file and partitioning the databases, which is described in “Scaling the Application Further” on page 2-22.

The sections that follow describe how the Production sample application uses replicated server processes and server groups, object state management, and factory-based routing to meet its scalability goals.

Changing the OMG IDL

The only OMG IDL changes for the Production sample application are limited to the `find_registrar()` and `find_teller()` operations on, respectively, the `RegistrarFactory` and `TellerFactory` objects. These two operations need to be modified to require, respectively, a student ID and account number, which are needed to implement factory-based routing. See “Scaling with Factory-based Routing” on page 2-11 to read about how the Production sample application implements and uses factory-based routing.

Using a Stateless Object Model

This section describes how object state management is used with the `Registrar` and `Teller` objects in the Production sample applications to increase the application’s scalability. For an introduction to object state management, see “Using Object State Management” on page 1-4.

To increase scalability, the `Registrar` and `Teller` objects are configured in the Production server application with the `method` activation policy. The `method` activation policy assigned to these two objects results in the following behavior changes:

- Whenever these objects are invoked, they are instantiated by the `WebLogic` Enterprise domain in the appropriate server group.
- After the invocation is complete, the `WebLogic` Enterprise domain deactivates these objects.

With the `Basic` through the `Wrapper` sample applications, the `Registrar` object was `process`-bound (`process` activation policy). All client requests on the `Registrar` object invariably went to the same object instance in the memory of the server machine. The `Basic` sample application design may be adequate for a small-scale deployment. However, as client application demands increase, client requests on the `Registrar` object eventually become queued, and response time drops.

However, when the `Registrar` and `Teller` objects are stateless (method activation policy), and the server processes that manage these objects are replicated, the `Registrar` and `Teller` objects can process multiple client requests in parallel. The only constraint on the number of simultaneous client requests that these objects can handle is the number of server processes that are available that can instantiate the `Registrar` and `Teller` objects. These stateless objects, thereby, make for more efficient use of machine resources and reduced client response time.

Most importantly, so that WebLogic Enterprise can instantiate copies of the `Registrar` and `Teller` objects in each of the replicated server processes, each copy of these objects must be unique. To make each instance of these objects unique, the factories for those objects must assign unique object IDs to them.

For the WebLogic Enterprise application to instantiate copies of the `Registrar` and `Teller` objects in each of the replicated server application processes, each copy of the `Registrar` and `Teller` objects have a unique object ID (OID). The factories that create these objects are responsible for assigning them unique OIDs. For information about generating unique object IDs, see *Creating CORBA C++ Server Applications*. For more information about other design considerations, see “Additional Design Considerations” on page 2-17.

Scaling by Replicating Server Processes and Server Groups

This topic includes the following sections:

- Replicating Server Processes in the Production Application
- Replicating Server Groups in the Production Application
- Configuring Replicated Server Processes and Groups in the Production Application

This topic describes how the Production sample application was scaled by replicating server processes and server groups. For an introduction to this topic, see “Replicating Server Processes and Server Groups” on page 1-9.

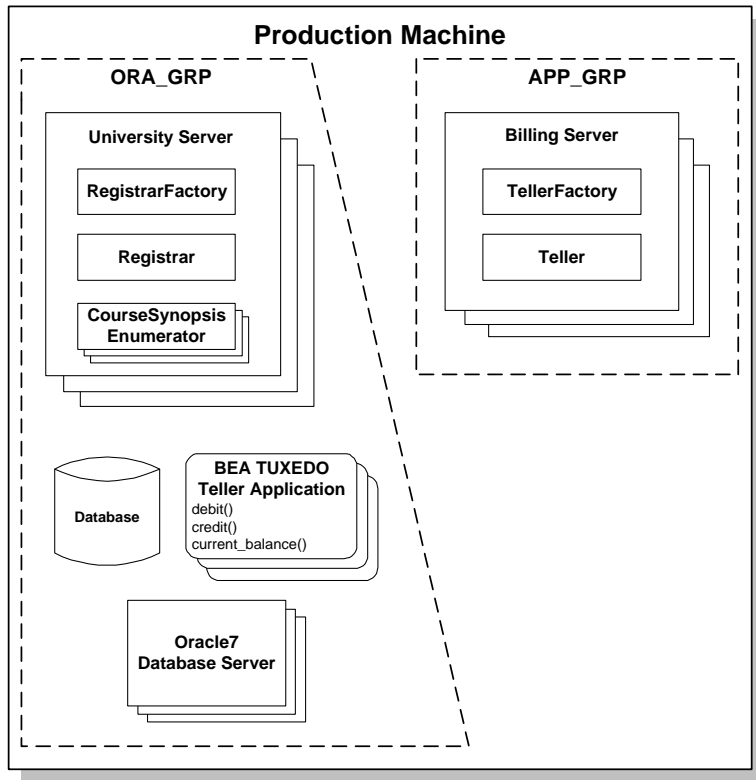
Replicating Server Processes in the Production Application

This section describes how the Production sample application replicates server applications. For an introduction to this feature, see “Replicating Server Processes” on page 1-11.

Figure 2-1 shows the replicated `ORA_GRP` and `APP_GRP` groups running on a single machine.

- The University server application, BEA Tuxedo Teller Application, and Oracle7 TMS server processes are replicated within the `ORA_GRP` group.
- The Billing server process is replicated within the `APP_GRP` group.

Figure 2-1 Replicated Server Groups in the Production Sample



When a request arrives for either of these groups, the WebLogic Enterprise domain has several server processes available that can process the request, and the WebLogic Enterprise domain can choose the server process that is the least busy.

In Figure 2-1, note the following points:

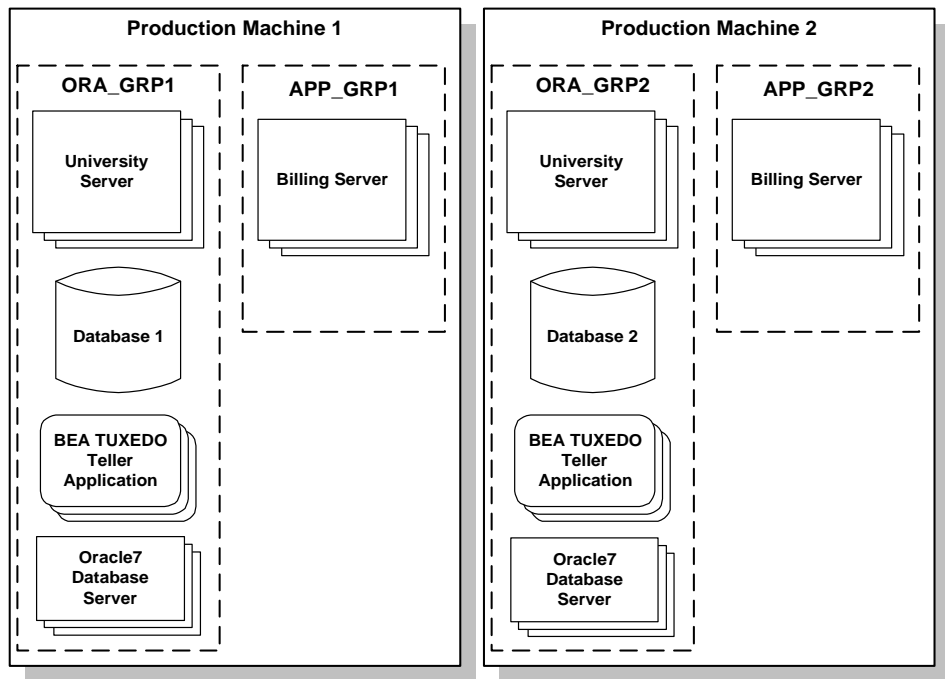
- At any time, there may be no more than one instance of the `RegistrarFactory`, `Registrar`, `TellerFactory`, or `Teller` objects within a given server process.
- There may be any number of `CourseSynopsisEnumerator` objects in any University server process.

Replicating Server Groups in the Production Application

This section describes how the Production sample application replicates server groups. For an introduction to this feature, see “Replicating Server Groups” on page 1-12.

Figure 2-2 shows the Production sample application groups replicated on another machine, as specified in the application’s `UBBCONFIG` file, as `ORA_GRP2` and `APP_GRP2`.

Figure 2-2 Replicating Server Groups Across Machines



In Figure 2-2, the only difference between the content of the groups on Production Machines 1 and 2 is the database:

- The database on Production Machine 1 contains student and account information for students with IDs between 100001 and 100005.

- The database on Production Machine 2 contains student and account information for students with IDs between 100006 and 100010.

Note: The course information table in both databases is identical.

Note that the student information in a given database may be completely unrelated to the account information in the same database.

For more information about how the Production sample application uses factory-based routing to distribute the application's processing load across multiple machines, see "Scaling with Factory-based Routing" on page 2-11.

Configuring Replicated Server Processes and Groups in the Production Application

Listing 2-1 shows excerpts from the GROUPS and SERVERS sections of the UBBCONFIG file for the Production sample application.

Listing 2-1 GROUPS and SERVERS Sections in a UBBCONFIG File

```
*GROUPS
  APP_GRP1
    LMID      = SITE1
    GRPNO     = 2
    TMSNAME   = TMS
  APP_GRP2
    LMID      = SITE1
    GRPNO     = 3
    TMSNAME   = TMS
  ORA_GRP1
    LMID      = SITE1
    GRPNO     = 4
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
  ORA_GRP2
    LMID      = SITE1
    GRPNO     = 5
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = " "
```

2 *Scaling CORBA C++ Server Applications*

```
TMSNAME      = "TMS_ORA"

*SERVERS
# By default, activate 2 instances of each server
# and allow the administrator to activate up to 5
# instances of each server
DEFAULT:
    MIN        = 2
    MAX        = 5
tellp_server
    SRVGRP     = ORA_GRP1
    SRVID      = 10
    RESTART    = N
tellp_server
    SRVGRP     = ORA_GRP2
    SRVID      = 10
    RESTART    = N

billp_server
    SRVGRP     = APP_GRP1
    SRVID      = 10
    RESTART    = N
billp_server
    SRVGRP     = APP_GRP2
    SRVID      = 10
    RESTART    = N
univp_server
    SRVGRP     = ORA_GRP1
    SRVID      = 20
    RESTART    = N
univp_server
    SRVGRP     = ORA_GRP2
    SRVID      = 20
    RESTART    = N
```

Scaling with Factory-based Routing

This topic includes the following sections:

- About Factory-based Routing in the Production Application
- Configuring Factory-based Routing in the UBBCONFIG File
- Implementing Factory-based Routing in a Factory
- What Happens at Run Time

This topic describes how the Production sample application was scaled using factory-based routing. For an introduction to factory-based routing, see “Using Factory-based Routing (CORBA only)” on page 1-16.

About Factory-based Routing in the Production Application

This section describes how the Production sample application uses a factory-based routing. For an introduction to this feature, see “Using Factory-based Routing (CORBA only)” on page 1-16.

You can use factory-based routing to expand the load-balancing and scalability features of WebLogic Enterprise. In the Production sample application, you can use factory-based routing to send requests to register one subset of students to one machine, and requests for another subset of students to another machine. As you increase your application’s processing capability, you can easily modify the factory-based routing in your application to add more machines.

The primary design consideration regarding implementing factory-based routing in the Production sample application is in choosing the value on which routing is based. The Production sample application uses factory-based routing in the following ways:

- Requests from client applications to the `Registrar` object are routed based on the student ID. Requests from student ID 100001 to 100005 go to Production Machine 1. Requests from student ID 100006 to 100010 go to Production Machine 2.

- Requests from the Registrar object to the Teller object are routed based on account number. Billing requests for account 200010 to 200014 go to Production Machine 1. Billing requests for account 200015 to 200019 go to Production Machine 2.

Configuring Factory-based Routing in the UBBCONFIG File

The University Production sample application demonstrates how to implement factory-based routing. The INTERFACES, ROUTING, and GROUPS sections from the ubb_b.nt configuration file show how you can implement factory-based routing in a WebLogic Enterprise application. You can find the ubb_p.nt or ubb_p.mk UBBCONFIG files for this sample in the directory where the WebLogic Enterprise software is installed (see the \samples\corba\university\production subdirectory).

The UBBCONFIG file must specify the following data in the INTERFACES and ROUTING sections, as well as how groups and machines are identified.

1. The INTERFACES section lists the names of the interfaces for which you want to enable factory-based routing. For each interface, this section specifies the kinds of criteria on which the interface routes. This section specifies the routing criteria via an identifier, FACTORYROUTING, as shown in Listing 2-2.

Listing 2-2 INTERFACES Section of a UBBCONFIG File

```
INTERFACES
  "IDL:beasys.com/UniversityP/Registrar:1.0"
    FACTORYROUTING = STU_ID
  "IDL:beasys.com/BillingP/Teller:1.0"
    FACTORYROUTING = ACT_NUM
```

Listing 2-2 shows the fully qualified interface names for the two interfaces in the Production sample in which factory-based routing is used. The FACTORYROUTING identifier specifies the names of the routing values, which are STU_ID and ACT_NUM, respectively.

2. The `ROUTING` section specifies the parameters in Table 2-1 for each routing value.

Table 2-1 Parameters Specified in the `ROUTING` Section

Parameter	Description
<code>TYPE</code>	Specifies the type of routing. In the Production sample, the type of routing is factory-based routing. Therefore, this parameter is defined as <code>FACTORY</code> .
<code>FIELD</code>	Specifies the variable name that the factory inserts in the routing value. In the Production sample, the field parameters are <code>student_id</code> and <code>account_number</code> , respectively.
<code>FIELDTYPE</code>	Specifies the data type of the routing value. In the Production sample, the field types for <code>student_id</code> and <code>account_number</code> are <code>long</code> .
<code>RANGES</code>	Specifies the values that are routed to each group.

Listing 2-3 shows the `ROUTING` section of the `UBBCONFIG` file used in the Production sample application.

Listing 2-3 `ROUTING` Section of the `UBBCONFIG` File

```
ROUTING
  STU_ID
    FIELD      = "student_id"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"
  ACT_NUM
    FIELD      = "account_number"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

Listing 2-3 shows that `Registrar` object references for students with IDs in one range are routed to one server group, and `Registrar` object references for students with IDs in another range are routed to another group. Likewise, `Teller` object references for accounts in one range are routed to one server

group, and `Teller` object references for accounts in another range are routed to another group.

3. The groups specified by the `RANGES` identifier in the `ROUTING` section of the `UBBCONFIG` file need to be identified and configured. For example, the Production sample specifies four groups: `APP_GRP1`, `APP_GRP2`, `ORA_GRP1`, and `ORA_GRP2`. These groups need to be configured, and the machines on which they run need to be identified.

Listing 2-4 shows the `GROUPS` section of the Production sample `UBBCONFIG` file, in which the `ORA_GRP1` and `ORA_GRP2` groups are configured. Notice how the names in the `GROUPS` section match the group names specified in the `ROUTING` section. This is critical for factory-based routing to work correctly. Furthermore, any change in the way groups are configured in an application must be reflected in the `ROUTING` section. (Note that the Production sample packaged with the WebLogic Enterprise software is configured to run entirely on one machine. However, you can easily configure this application to run on multiple machines.)

Listing 2-4 GROUPS Section of a UBBCONFIG File

```
*GROUPS
  APP_GRP1
    LMID      = SITE1
    GRPNO     = 2
    TMSNAME   = TMS
  APP_GRP2
    LMID      = SITE1
    GRPNO     = 3
    TMSNAME   = TMS
  ORA_GRP1
    LMID      = SITE1
    GRPNO     = 4
    OPENINFO  =
"ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=."+MaxCur=5"
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
  ORA_GRP2
    LMID      = SITE1
    GRPNO     = 5
OPENINFO = "ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=."+MaxCur=5"
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
```

Implementing Factory-based Routing in a Factory

Factories implement factory-based routing in the way the invocation to the `TP::create_object_reference()` operation is implemented. This operation has the C++ binding in Listing 2-5.

Listing 2-5 C++ Binding for `create_object_reference`

```
CORBA::Object_ptr  TP::create_object_reference (
                    const char* interfaceName,
                    const PortableServer::oid &stroid,
                    CORBA::NVlist_ptr criteria);
```

The third parameter to this operation, `criteria`, specifies a list of named values to be used for factory-based routing. To implement factory-based routing in a factory, you need to build the `NVlist`. The use of factory-based routing is optional and is dependent on this argument. Instead of using factory-based routing, you can pass a value of 0 (zero) for this argument.

As stated previously, the `RegistrarFactory` object in the Production sample application specifies the value `STU_ID`. This value must exactly match the following information in the `UBBCONFIG` file:

- The routing name, type, and allowable values specified by the `FACTORYROUTING` identifier in the `INTERFACES` section.
- The routing criteria name, field, and field type specified in the `ROUTING` section.

The `RegistrarFactory` object inserts the student ID into the `NVlist` using the code shown in Listing 2-6.

Listing 2-6 NVlist in the RegistrarFactory Object

```
// put the student id (which is the routing criteria)
// into a CORBA NVList:
CORBA::NVlist_var v_criteria;
TP::orb()->create_list(1, v_criteria.out());
CORBA::Any any;
```

```
any <= (CORBA::Long)student;  
v_criteria->add_value("student_id", any, 0);
```

The RegistrarFactory object has the invocation to the TP::create_object_reference() operation, shown in Listing 2-7, passing the NVlist created in Listing 2-6.

Listing 2-7 Invoking create_object_reference in the RegistrarFactory Object

```
// create the registrar object reference using  
// the routing criteria :  
CORBA::Object_var v_reg_oref =  
    TP::create_object_reference(  
        UniversityP::_tc_Registrar->id(),  
        object_id,  
        v_criteria.in()  
    );
```

The Production sample application also uses factory-based routing in the TellerFactory object to determine the group in which Teller objects should be instantiated based on an account number.

What Happens at Run Time

When you implement factory-based routing in a factory, WebLogic Enterprise generates an object reference. The following example shows how the client application gets an object reference to a Registrar object when factory-based routing is implemented.

1. The client application invokes the RegistrarFactory object, requesting a reference to a Registrar object. The request includes a student ID.
2. The RegistrarFactory inserts the student ID into an NVlist, which is used as the routing criteria.

3. The `RegistrarFactory` invokes the `TP::create_object_reference()` operation, passing the `Registrar` interface name, a unique OID, and the `NVlist`.
4. WebLogic Enterprise compares the contents of the routing tables with the value in the `NVlist` to determine a group ID.
5. WebLogic Enterprise inserts information about the group into the object reference.

When the client application subsequently invokes an object using the object reference, WebLogic Enterprise routes the request to the group specified in the object reference.

Note: If you use the process-entity design pattern, you should use caution in how you implement factory-based routing. The object can service only those entities that are contained in the group's database.

Additional Design Considerations

This topic includes the following sections:

- About the Additional Design Considerations
- Instantiating the Registrar and Teller Objects
- Ensuring That Student Registration Occurs in the Correct Server Group
- Ensuring That the Teller Object Is Instantiated in the Correct Server Group

About the Additional Design Considerations

When designing the `Registrar` and `Teller` objects, you should ensure that:

- The `Registrar` and `Teller` objects work properly for the Production deployment environment; namely, across multiple replicated server processes and multiple groups. Given that the University and Billing server processes are replicated, the design must consider how these two objects should be instantiated.

- Client requests for registration and billing operations for a given student go to the correct server group, given that the two server groups in the Production WebLogic Enterprise domain each deal with different databases.

These objects must have unique object IDs (OIDs) and must be method-bound (that is, they must have the method activation policy assigned to them).

Instantiating the Registrar and Teller Objects

In the University server applications that are less sophisticated than the Production sample application, the run-time behavior of the `Registrar` and `Teller` objects was simpler:

- Each object was process-bound, meaning that each was activated the first time it was invoked, and it stayed in memory until the server process in which it ran was shut down.
- Since there was only one server group running in the WebLogic Enterprise domain, and only one University and Billing server process in the group, all client requests were directed to the same objects. As multiple client requests arrived in the WebLogic Enterprise domain, these objects each processed one client request at one time.
- Because there was only one instance of each object in the server processes in which they ran, neither object needed a unique OID. The OID for each object specified only the Interface Repository ID.

However, because the University and Billing server processes are now replicated, WebLogic Enterprise must be able to differentiate among multiple instances of the `Registrar` and `Teller` objects. For example, if there are two University server processes running in a group, WebLogic Enterprise must have a means to distinguish between the `Registrar` object running in the first University server process and the `Registrar` object running in the second University server process. To distinguish multiple instances of these objects, each object instance must be unique.

To make each `Registrar` and `Teller` object unique, the factories for those objects must change the way in which they make object references to them. For example, when the `RegistrarFactory` object in the Basic sample application created an object reference to the `Registrar` object, the `TP::create_object_reference()`

operation specified an OID that consisted only of the string `registrar`. However, in the Production sample application, the same `TP::create_object_reference()` operation uses a generated unique OID instead.

As a result of giving each `Registrar` and `Teller` object a unique OID, multiple instances of these objects may be running simultaneously in the WebLogic Enterprise domain. This characteristic is typical of the stateless object model, and is an example of how the WebLogic Enterprise domain can be highly scalable while it offers high performance.

Finally, because unique `Registrar` and `Teller` objects need to be brought into memory for each client request on them, it is critical that these objects be deactivated when the invocations on them are completed so that any object state associated with them does not remain idle in memory. The Production server application addresses this issue by assigning the `method` activation policy to these two objects in the Implementation Configuration File (ICF).

Ensuring That Student Registration Occurs in the Correct Server Group

The primary scalability advantage of using replicated server groups is being able to distribute processing across multiple machines. However, if your application interacts with a database, which is the case with the University sample applications, it is critical that you consider the impact of these multiple server groups on the database interactions.

In many cases, you may have one database associated with each machine in your deployment. If your server application is distributed across multiple machines, you must consider how you set up your databases.

The Production sample application, as described in this chapter, uses two databases. However, this application can easily be configured to accommodate more. The system administrator can decide on how many databases to use.

In the Production sample application, the student and account information is partitioned across the two databases, but course information is identical. Having identical course information in both databases is not a problem because the course information is read-only for the purposes of course registration. However, the student and account information is read-write. If multiple databases were also to contain

identical data for students and accounts (that is, the database is not partitioned), the application would need to deal with the overhead of synchronizing the updates to student and account information across all the databases each time any student or account information were to change.

The Production sample application uses factory-based routing to send one set of requests to one machine, and another set to the other machine. How factory-based routing is implemented in the `RegistrarFactory` object depends on the way in which references to `Registrar` objects are created.

For example, when the client application sends a request to the `RegistrarFactory` object to get an object reference to a `Registrar` object, the client application includes a student ID in that request. The client application must use the object reference that the `RegistrarFactory` object returns to make all subsequent invocations on a `Registrar` object on a particular student's behalf, because the object reference returned by the factory is group-specific. Therefore, for example, when the client application subsequently invokes the `get_student_details()` operation on the `Registrar` object, the client application can be assured that the `Registrar` object is active in the server group associated with the database containing data for that student.

To show how this works, consider the following execution scenario, which is implemented in the Production sample application:

1. The client application invokes the `find_registrar()` operation on the `RegistrarFactory` object. Included in this invocation is the student ID 1000003.
2. WebLogic Enterprise routes the client request to any `RegistrarFactory` object.
3. The `RegistrarFactory` object uses the student ID to create an object reference to a `Registrar` object in `ORA_GRP1`, based on the routing information in the `UBBCONFIG` file, and returns that object reference to the client application.
4. The client application invokes the `register_for_courses()` operation on the `Registrar` object.
5. WebLogic Enterprise receives the client request and routes it to the server group specified in the object reference. In this case, the client request goes to the University server process in `ORA_GRP1`, which is on Production Machine 1.
6. The University server process instantiates a `Registrar` object and sends the client invocation to it.

The RegistrarFactory object from the preceding scenario returns to the client application a unique reference to a Registrar object that can be instantiated only in ORA_GRP1, which runs on Production Machine 1 and has a database containing student data for students with IDs in the range 100001 to 100005. Therefore, when the client application sends subsequent requests to this Registrar object on behalf of a given student, the Registrar object interacts with the correct database.

Ensuring That the Teller Object Is Instantiated in the Correct Server Group

When the Registrar object needs a Teller object, the Registrar object invokes the TellerFactory object, using the TellerFactory object reference cached in the University Server object.

However, because factory-based routing is used in the TellerFactory object, the Registrar object passes the student's account number when the Registrar object requests a reference to a Teller object. This way, the TellerFactory object creates a reference to a Teller object in the group that has the correct database.

Note: For the Production sample application to work properly, it is essential that the system administrator configures the server groups and the databases properly. In particular, the system administrator must make sure that a match exists between the routing criteria specified in the routing tables and the databases to which requests using those criteria are routed. Using the Production sample as an example, the database in a given group must contain the correct student and account information for the requests that are routed to that group.

Scaling the Application Further

In the future, the system administrator of the Production sample application may want to add capacity to the WebLogic Enterprise domain. For example, the University may eventually experience a large increase in the student population, or the Production application may be scaled up to accommodate the course registration process for an entire state university system, encompassing several campuses. This can be done without modifying or rebuilding the application.

The system administrator can continually add capacity by:

- Replicating the server groups in the Production sample application across additional machines.

The system administrator must modify the `UBBCONFIG` file to specify the additional server groups, the server processes that run in those groups, and the machines on which the server groups run.

- Changing the factory-based routing tables.

For example, instead of routing to the two existing groups in the Production sample application, the system administrator can modify the routing rules in the `UBBCONFIG` file to partition the application further among additional server groups added to the WebLogic Enterprise domain. Any modification to the routing tables must match the information for the configured server groups and machines in the `UBBCONFIG` file.

Note: If you add capacity to an existing WebLogic Enterprise application that uses a database, you must also consider the impact on how the database is set up, particularly when you are using factory-based routing. For example, if the Production sample application is distributed across six machines, the database on each machine must be set up appropriately and in accordance with the routing tables in the `UBBCONFIG` file.

3 Scaling CORBA Java Server Applications

Using the JDBC Bankapp sample application as an example, this topic demonstrates scaling a WebLogic Enterprise CORBA Java application to increase its processing capability. This topic includes the following sections:

- About Scaling the JDBC Bankapp Sample Application
- Scaling with Object State Management
- Scaling by Replicating Server Processes and Server Groups
- Scaling with Factory-based Routing
- Additional Design Considerations
- Scaling the Application Further

Before you begin, be sure to read Chapter 1, “Scaling WebLogic Enterprise Applications,” for a comprehensive introduction to tuning and scaling WebLogic Enterprise applications. For information about building and running the JDBC Bankapp sample application, see the *Bankapp Sample Using JDBC* in the WebLogic Enterprise online documentation.

Note: Some of the Bankapp examples in this topic include sample code that is *not* implemented in the sample Bankapp files that ship with WebLogic Enterprise.

About Scaling the JDBC Bankapp Sample Application

This topic includes the following sections:

- Design Goals
- How the Application Has Been Scaled

Design Goals

The primary design goal of the JDBC Bankapp sample application is to significantly increase the number of client applications it can accommodate by:

- Processing in parallel, and on one machine, client requests on multiple objects that implement the same interface.
- Directing requests on behalf of certain bank automated teller machines (ATMs) to one machine, and other ATMs to other machines.
- Adding more machines to share the processing load.

How the Application Has Been Scaled

To accommodate these design goals, the JDBC Bankapp sample application has been scaled by:

- Implementing a stateless object model to scale up the number of client requests the server process can manage simultaneously.
- Replicating the `Teller` and `TellerFactory` server processes within the groups in which they are configured.
- Replicating the groups described previously on an additional machine.

- Assigning unique object IDs (OIDs) to the following objects so that they can be instantiated multiple times simultaneously in their respective groups:

- `TellerFactory`
- `Teller`

This makes these objects available on a per-client application (and not per-process) basis, thereby accommodating a parallel processing capability.

- Implementing factory-based routing to direct client requests on behalf of some ATMs to one machine, and other ATMs to another machine.
- Setting up threads for the `Teller` object. For related information, also see “Using Multithreaded Java Servers (Java only)” on page 1-13.

The sections that follow describe how the JDBC Bankapp sample application uses replicated server processes and server groups, object state management, and factory-based routing to meet its scalability goals.

Scaling with Object State Management

This section describes how object state management is used with the `Teller` objects in the Bankapp sample application to increase the application’s scalability. For an introduction to object state management, see “Using Object State Management” on page 1-4.

For example, the Bankapp sample `Teller` object could use the `method` activation policy. The `method` activation policy assigned to this object means that the object is activated whenever a client request arrives for it. The `Teller` object remains in memory only for the duration of one client invocation, which is appropriate in cases where the Process-Entity design pattern is recommended. For more information about the Process-Entity design pattern, see the technical article *Process-Entity Design Pattern*.

As the number of clients issuing requests on the `Teller` object increases, WebLogic Enterprise can:

- Instantiate the `Teller` object for each client request that arrives. Client requests are not queued for an existing `Teller` object, which would likely be the case if the `Teller` object were process-bound.

- Perform load balancing by instantiating the `Teller` object on the least busy server process or group.

Scaling by Replicating Server Processes and Server Groups

This topic includes the following sections:

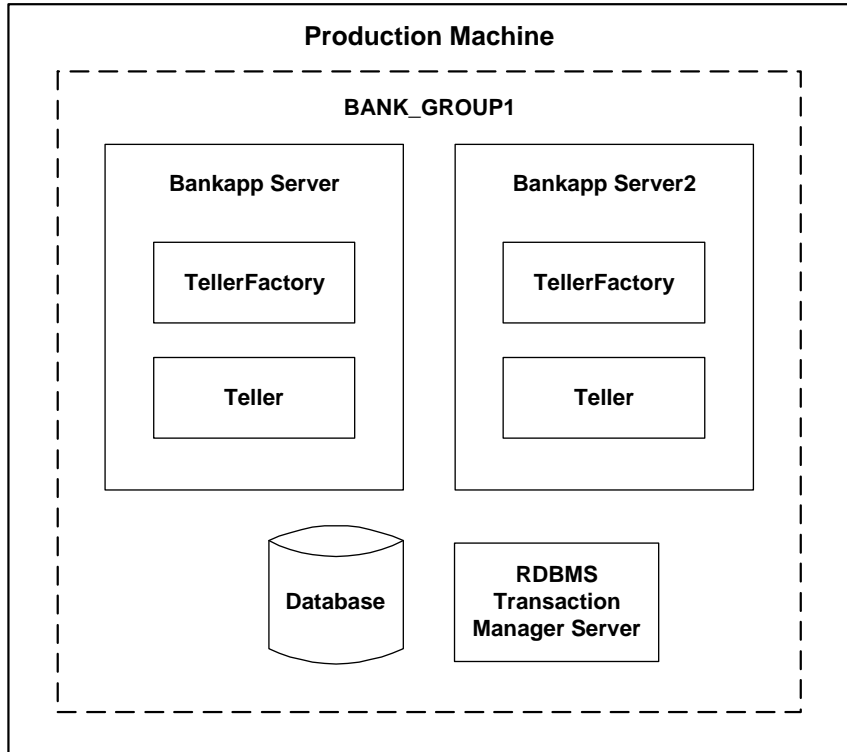
- Replicating Server Processes in the Bankapp Application
- Replicating Server Groups in the Bankapp Application
- Configuring Replicated Server Processes and Groups in the Bankapp Application

This topic describes how the BankApp server application was scaled by replicating server processes and server groups. For an introduction to this topic, see “Replicating Server Processes and Server Groups” on page 1-9.

Replicating Server Processes in the Bankapp Application

Figure 3-1 shows the Bankapp server application replicated in the `BANK_GROUP1` group. The replicated servers are running on a single machine.

Figure 3-1 Replicated Servers in the Bankapp Sample



When a request arrives for this group, WebLogic Enterprise has several server processes available that can process the request, and WebLogic Enterprise can choose the server process that is the least busy.

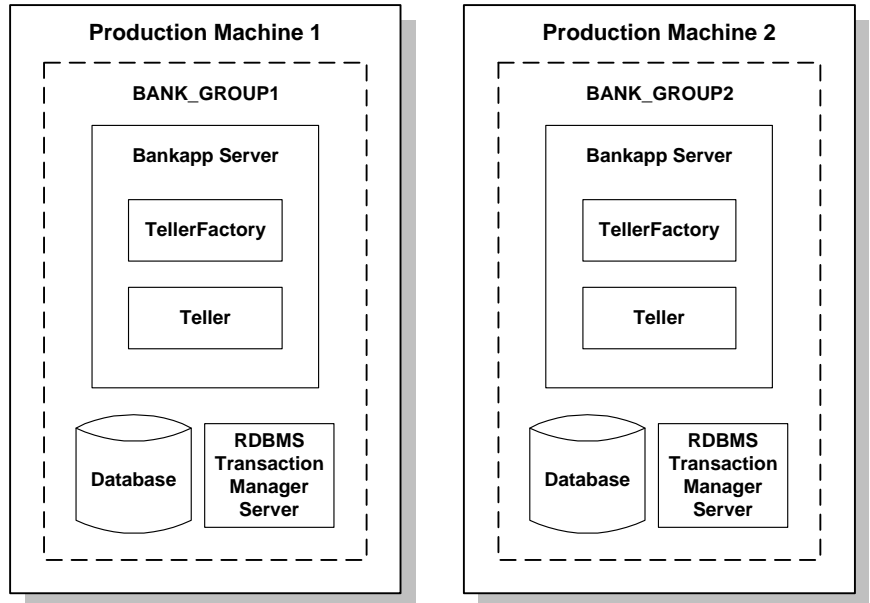
In Figure 3-1, note the following:

- At any time, there may be no more than one instance of the `TellerFactory` object within a given server process.
- There may be any number of `Teller` objects in any Bankapp server process.

Replicating Server Groups in the Bankapp Application

Figure 3-2 shows the Bankapp sample application groups replicated on another machine, as specified in the application's `UBBCONFIG` file.

Figure 3-2 Replicating Server Groups Across Machines



Note: In the simple example shown in Figure 3-2, the content of the databases on Production Machines 1 and 2 is identical. Each database contains all of the account records for all of the account IDs. Only the processing is distributed, based on the ATM (`atmID` field). A more realistic example would distribute the data and processing based on ranges of bank account IDs.

For more information about how the Bankapp sample application uses factory-based routing to distribute the application's processing load across multiple machines, see "Scaling with Factory-based Routing" on page 3-10.

Configuring Replicated Server Processes and Groups in the Bankapp Application

Listing 3-1 shows excerpts from the GROUPS and SERVERS sections of the UBBCONFIG file for a Bankapp sample application.

Note: These configuration settings are *not* used with the Bankapp sample provided with the WebLogic Enterprise software.

Listing 3-1 Excerpts from GROUPS and SERVERS Section of UBBCONFIG

```
*RESOURCES
    IPCKEY          55432
    DOMAINID        simple
    MASTER          SITE1
    MODEL           SHM
    LDBAL           Y

*MACHINES
    "TRIXIE"
        LMID        = SITE1
        APPDIR       = "c:\bankapp\jdbc\"
        TUXCONFIG    = "c:\bankapp\jdbc\.\tuxconfig"
        TUXDIR       = "c:\m3dir"
        MAXCLIENTS  = 10

*GROUPS
    SYS_GRP
        LMID        = SITE1
        GRPNO       = 1
    BANK_GROUP1
        LMID        = SITE1
        GRPNO       = 2
    BANK_GROUP2
        LMID        = SITE1
        GRPNO       = 3

*SERVERS
    # By default, restart a server if it crashes, up to 5 times
    # in 24 hours.
    #
    DEFAULT:
        RESTART     = Y
        MAXGEN      = 5
```

3 *Scaling CORBA Java Server Applications*

```
# Start the Tuxedo System Event Broker. This event broker
# must be started before any servers providing the
# NameManager Service.
#
TMSYSEVT
    SRVGRP = SYS_GRP
    SRVID = 1

# TMFFNAME is a M3 provided server that runs the
# object-transactional management services. This includes the
# NameManager and FactoryFinder services.

# The NameManager service is a M3-specific service
# that maintains a mapping of application-supplied names to
# object references.

# Start the NameManager Service (-N option). This name
# manager is being started as a Master (-M option).
#

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 2
    CLOPT = "-A -- -N -M"

# Start a slave NameManager Service
#

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 3
    CLOPT = "-A -- -N"

# Start the FactoryFinder (-F) service
#

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 4
    CLOPT = "-A -- -N -F"

# Start the JavaServer in Bank_Group1
#
JavaServer
    SRVGRP = BANK_GROUP1
    SRVID = 5
    CLOPT = "-A -- -M 10 BankApp.jar TellerFactory_1"
    SYSTEM_ACCESS=FASTPATH
```



```

                                RESTART = N

# Start the JavaServer in Bank_Group2
#

    JavaServer
        SRVGRP = BANK_GROUP2
        SRVID = 6
        CLOPT = "-A -- -M 10 BankApp.jar TellerFactory_1"
        SYSTEM_ACCESS=FASTPATH
        RESTART = N

# Start the listener for IIOP clients
#
# Specify the host name of your server machine as
# well as the port. A typical port number is 2500
#

    ISL
        SRVGRP = SYS_GRP
        SRVID = 7
        CLOPT = "-A -- -n //TRIXIE:2468"

*SERVICES

*INTERFACES
    "IDL:beasys.com/BankApp/Teller:1.0"
    FACTORYROUTING=atmID

*ROUTING
    atmID
        TYPE = FACTORY
        FIELD = "atmID"
        FIELDTYPE = LONG
        RANGES = "1-5:BANK_GROUP1,
                  6-10: BANK_GROUP2,
                  *:BANK_GROUP1"
```

Scaling with Factory-based Routing

This topic includes the following sections:

- About Factory-based Routing in the Bankapp Application
- Configuring Factory-based Routing in the UBBCONFIG File
- Implementing Factory-based Routing in a Factory
- What Happens at Run Time

This topic describes how the BankApp server application was scaled using factory-based routing. For an introduction to factory-based routing, see “Using Factory-based Routing (CORBA only)” on page 1-16.

About Factory-based Routing in the Bankapp Application

You can use factory-based routing to expand the load-balancing and scalability features of WebLogic Enterprise. In the Bankapp sample application, you can use factory-based routing to send requests to a subset of ATMs to one machine, and requests for another subset of ATMs to another machine. As you increase your application’s processing capability, you can easily modify the factory-based routing in your application to add more machines.

The primary design consideration regarding implementing factory-based routing in the Bankapp sample application is in choosing the value on which routing is based. The following sections describe how factory-based routing works in the JDBC Bankapp sample application. Client application requests to the `Teller` object are routed based on a teller number:

- Requests for one subset of teller numbers are routed to one group.
- Requests on behalf of another subset of teller numbers are routed to another group.

Configuring Factory-based Routing in the UBBCONFIG File

The UBBCONFIG file must specify the following data in the `INTERFACES` and `ROUTING` sections, as well as how groups and machines are identified.

1. The `INTERFACES` section lists the names of the interfaces for which you want to enable factory-based routing. For each interface, this section specifies the kinds of criteria on which the interface routes. This section specifies the routing criteria via an identifier, `FACTORYROUTING`, as shown in Listing 3-2.

Listing 3-2 Sample INTERFACES Section

```
*INTERFACES
"IDL:beasys.com/BankApp/Teller:1.0"
    FACTORYROUTING = atmID
```

Listing 3-2 shows the fully qualified Interface Repository ID for an interface in the extended Bankapp sample in which factory-based routing is used. The `FACTORYROUTING` identifier specifies the name of the routing value, `atmID`.

2. The `ROUTING` section specifies the parameters in Table 3-1 for each routing value.

Table 3-1 Parameters Specified in the ROUTING Section

Parameter	Description
TYPE	Specifies the type of routing. In the Bankapp sample, the type of routing is factory-based routing. Therefore, this parameter is defined as <code>FACTORY</code> .
FIELD	Specifies the name that the factory inserts in the routing value. In the extended Bankapp sample, the field parameter is <code>atmID</code> .
FIELDTYPE	Specifies the data type of the routing value. In the Bankapp sample, the field type for <code>atmID</code> is <code>LONG</code> .
RANGES	Specifies the values that are routed to each group.

Listing 3-3 shows the ROUTING section of the UBBCONFIG file used in the Bankapp sample application.

Listing 3-3 Sample ROUTING Section

```
*ROUTING
  atmID
      TYPE = FACTORY
      FIELD = "atmID"
      FIELDTYPE = LONG
      RANGES = "1-5: BANK_GROUP1,
                6-10: BANK_GROUP2,
                *:BANK_GROUP1"
```

Listing 3-3 shows that `Teller` object references for ATMs in one range are routed to one server group, and `Teller` object references for ATMs in other ranges are routed to other groups. As shown in Figure 3-2, `BANK_GROUP1` and `BANK_GROUP2` reside on different production machines.

Implementing Factory-based Routing in a Factory

Factories implement factory-based routing in the way in which the invocation to the `com.beasys.Tobj.TP.create_object_reference` method is implemented.

Listing 3-4 shows the Java binding for this operation.

Listing 3-4 Java Binding for `create_object_reference`

```
public static org.omg.CORBA.Object
  create_object_reference(java.lang.String interfaceName,
                        java.lang.String stroid,
                        org.omg.CORBA.NVList criteria)
    throws InvalidInterface,
           InvalidObjectId
```

The `criteria` specifies a list of named values that can be used to provide factory-based routing for the object reference. The use of factory-based routing is optional and is dependent on this argument. Instead of using factory-based routing, you can pass a value of 0 (zero) for this argument. To implement factory-based routing in a factory, you need to build the `NVlist`.

As stated previously, the `TellerFactory` object in the `Bankapp` sample application specifies the value `atmID`. This value must exactly match the following information in the `UBBCONFIG` file:

- The routing name, type, and allowable values specified by the `FACTORYROUTING` identifier in the `INTERFACES` section.
- The routing criteria name, field, and field type specified in the `ROUTING` section.

Note: Listing 3-5 is *not* part of the `Bankapp` sample code, but is included here to illustrate factory-based routing. The `TellerFactory` object inserts the bank account number into the `NVlist` using the following code.

Listing 3-5 Sample of Factory-Based Routing

```
// Put the atmID (which is the routing criteria)
// into a CORBA NVList. The atmID comes from the
// tellerName that is passed in as an input parameter;
// tellerName should have the form: Teller<atmID>

int atmID = Integer.parseInt (tellerName.substring(6));
any.insert_long(atmID);

// Create the NVlist and add the atmID to the list.

org.omg.CORBA.NVlist criteria = TP.orb().create_list(1);
criteria.add_value("atmID", any, 0);

// Create the object reference.

    org.omg.CORBA.Object teller_oref =
        TP.create_object_reference(
            BankApp.TellerHelper.id(), // Repository ID
            tellerName,                 // Object ID
            criteria                     // Routing Criteria
        );
```

What Happens at Run Time

When you implement factory-based routing in a factory, WebLogic Enterprise generates an object reference. The following example shows how the client application gets an object reference to a `Teller` object when factory-based routing is implemented:

1. The client application invokes the `TellerFactory` object, requesting a reference to a `Teller` object. The request includes a teller name, which includes an `atmID`.
2. The `TellerFactory` inserts the `atmID` into an `NVlist`, which is used as the routing criteria.
3. The `TellerFactory` invokes the `com.beasys.Tobj.TP::create_object_reference` method, passing the `Teller` Interface Repository ID, a unique OID, and the `NVlist`.
4. WebLogic Enterprise compares the content of the routing tables with the value in the `NVlist` to determine a group ID.
5. WebLogic Enterprise inserts the group ID into the object reference.

When the client application subsequently invokes an object using the object reference, WebLogic Enterprise routes the request to the group specified in the object reference.

Note: If you use the process-entity design pattern, you should use caution in how you implement factory-based routing. The object can service only those entities that are contained in the group's database.

Additional Design Considerations

This topic includes the following sections:

- About the Additional Design Considerations
- Instantiating the Teller Object
- Ensuring That Account Updates Occur in the Correct Server Group

About the Additional Design Considerations

When designing the `Teller` object, you should ensure that:

- The `Teller` object works properly for the Bankapp deployment environment; namely, across multiple replicated server processes and multiple groups.
- Client requests for account inquiries, withdrawals, and transfers in a given account go to the correct server group, given that the four server groups in the extended Bankapp WebLogic Enterprise domain each interact with different databases.

These objects must have unique object IDs (OIDs) and must be method-bound (that is, they must have the `method` activation policy assigned to them).

Instantiating the Teller Object

Because the extended Bankapp server is now replicated, the WebLogic Enterprise domain must be able to differentiate among multiple instances of the `Teller` object. For example, if there are two Bankapp server processes running in a group, WebLogic Enterprise must be able to distinguish between a `Teller` object running in the first Bankapp server process and a `Teller` object running in the second Bankapp server process. To distinguish multiple instances of these objects, each object instance must be unique.

To make each `Teller` object unique, the factories for those objects must change the way in which they make object references to them. For example, when the `TellerFactory` object in the original Bankapp sample application created an object reference to the `Teller` object, the `com.beasys.Tobj.TP::create_object_reference` method specified an OID that consisted only of the string `tellerName`. However, in the extended Bankapp sample application discussed in this chapter, the same `create_object_reference` method uses a generated unique OID instead.

As a result of giving each `Teller` object a unique OID, multiple instances of these objects may be running simultaneously in the WebLogic Enterprise domain. This characteristic is typical of the stateless object model, and is an example of how the WebLogic Enterprise domain can be highly scalable while it offers high performance.

Finally, because unique `Teller` objects need to be brought into memory for each client request on them, it is critical that these objects be deactivated when the invocations on them are completed so that any object state associated with them does not remain idle in memory. The Bankapp server application addresses this issue by assigning the method activation policy to the `Teller` object in the XML-based Server Description File.

Ensuring That Account Updates Occur in the Correct Server Group

The primary scalability advantage of using replicated server groups is being able to distribute processing across multiple machines. However, if your application interacts with a database, which is the case with the JDBC Bankapp sample application, it is critical that you consider the impact of these multiple server groups on the database interactions.

In many cases, you may have one database associated with each machine in your deployment. If your server application is distributed across multiple machines, you must consider how you set up your databases.

The JDBC Bankapp sample application uses factory-based routing to send one set of requests to one machine, and another set to the other machine. How factory-based routing is implemented in the `TellerFactory` object depends on how references to `Teller` objects are created.

Scaling the Application Further

In the future, the system administrator of the Bankapp sample application may want to add capacity to the WebLogic Enterprise domain. For example, the bank may eventually have a large increase in automated teller machines (ATMs). This can be done without modifying or rebuilding the application.

The system administrator can continually add capacity by:

- Replicating the Bankapp sample application server groups across additional machines.

The system administrator must modify the `UBBCONFIG` file to specify the additional groups, the server processes that run in those groups, and the machines on which they run.

- Changing the factory-based routing tables.

For example, instead of routing to the four groups shown earlier in this chapter, the system administrator can modify the routing rules in the `UBBCONFIG` file to partition the application further among the new groups added to the WebLogic Enterprise domain. Any modification to the routing tables must be consistent with any changes or additions made to the server groups and machines configured in the `UBBCONFIG` file.

Note: If you add capacity to an application that uses a database, you must also consider the impact on how the database is set up, particularly when you are using factory-based routing. For example, if the Bankapp sample application is distributed across six machines, the database on each machine must be set up appropriately and in accordance with the routing tables in the `UBBCONFIG` file.

4 Scaling EJB Applications

This topic describes the EJB application scaling tasks associated with the EJB architecture roles specified in Chapter 3 of the Enterprise JavaBeans Specification 1.1, published by Sun Microsystems, Inc. The WebLogic Enterprise JavaServer provides an implementation of the EJB container as defined in this specification.

This topic includes the following sections:

- Scaling Tasks for EJB Providers
- Scaling Tasks for Application Assemblers and Deployers
- Scaling Tasks for System Administrators

Before you begin, be sure to read Chapter 1, “Scaling WebLogic Enterprise Applications,” for a comprehensive introduction to tuning and scaling WebLogic Enterprise applications. The concepts in that chapter apply to EJB applications as well. The main difference is that factory-based routing is *not* supported in EJB applications.

In addition, for an introduction to using EJB applications in the WebLogic Enterprise environment, see “Developing WebLogic Enterprise EJB Applications” in *Getting Started*.

Scaling Tasks for EJB Providers

This topic includes the following sections:

- Using Stateless Session Beans
- Minimizing State Information in Stateful Session Beans
- Using Pooled Connections
- Implementing Methods for Bean Persistence
- Completing Transactions Efficiently
- Implementing the Process-entity Design Pattern

For a general discussion about using stateful and stateless objects, see “Using Object State Management” on page 1-4.

Using Stateless Session Beans

EJB Providers can increase application scalability by using stateless session beans wherever appropriate. With stateless session beans, the WebLogic Enterprise EJB container can freely pool instances, allocate instances as needed, and apply load balancing strategies to distribute the load across different servers within the domain.

Stateless session beans can be load balanced on a per-request basis. With every method invocation, a stateless session bean can be relocated to the least busy server within a group or *across* groups within a domain. For more information about stateless session beans, see “Types of Beans Supported in WebLogic Enterprise” in “The WebLogic Enterprise JavaBeans Programming Environment” topic in *Getting Started*.

Although stateless session beans by definition do not have a persistent state for client applications, they *can* have instance variables that retain values while the EJB remains in an instance pool on the server. This WebLogic Enterprise feature allows EJB Providers to implement highly scalable EJBs. For example, suppose a stateless session bean invokes methods on a CORBA object. Normally, for each method invocation, the EJB needs to make two calls: one to obtain an object reference, and another to invoke the method. However, the EJB can store the object reference in an instance variable.

Thereafter, when an application invokes a method on the EJB, it can be retrieved from the pool if it is available in the pool. The application can then check the instance variable for a valid value; if it is valid, the application can avoid making an extra call to obtain the object reference.

Minimizing State Information in Stateful Session Beans

EJB Providers can increase application scalability by minimizing, in stateful session beans, the state information that must be stored and retrieved during passivation and reactivation.

Stateful objects (stateful session beans and entity beans) are generally more resource intensive than stateless objects because they allocate and exclusively reserve resources during the private conversation with the client. After the state is allocated for an object, the object remains linked to that server for the duration of the method invocation or the transaction. Stateful session beans can be load balanced using any server that supports the bean *within* the group only (not across groups).

Using Pooled Connections

EJB Providers can increase application scalability by using pooled database connections. The JDBC connection pool optimizes performance for database connections by reducing the overhead associated with opening each connection. For more information about configuring and using JDBC connection pools, see *Using JDBC Connection Pooling*.

Implementing Methods for Bean Persistence

To optimize application performance, the WebLogic Enterprise EJB container manages the passivation and reactivation of stateful objects (stateful session beans and entity beans) automatically, based on available system resources. The container can pool instances of a bean and decide when an instance can be removed from the pool to provide a more efficient use of system resources. The WebLogic Enterprise EJB container may passivate an object after a method invocation.

Note: An object will not be passivated while it participates in a transaction. The WebLogic Enterprise EJB container may passivate it only after the transaction is completed.

The WebLogic Enterprise EJB container manages load balancing with passivated objects. After it is passivated, the WebLogic Enterprise EJB container can relocate an object to the least busy server within the group as long as the object is idle (there are no pending requests on that object). This is particularly important when the bean accesses a database using cursors, because these cursors could become invalid after the passivation (the EJB container can reactivate the bean on a different server).

For stateful session beans and entity beans, EJB Providers can increase application scalability by implementing the `ejbPassivate` and `ejbActivate` methods in an efficient manner. For more information about persistence in EJB applications, see “EJBs and Persistence” in “The WebLogic Enterprise JavaBeans Programming Environment” topic in *Getting Started*.

Finally, for stateful session beans with container-managed persistence, EJB Providers should favor using JDBC-based persistence over file-based persistence. File-based persistence is generally less scalable. If the client process crashes (for example, the network connection is lost or the client machine is turned off), the file is not automatically removed. An accumulation of these files can slow performance.

Completing Transactions Efficiently

EJB Providers can increase application scalability by completing transactions efficiently. An object cannot be passivated while it is participating in a transaction. For example, EJB Providers can specify the timeout period for transactions in EJB applications. If the duration of a transaction exceeds the specified timeout setting, then the Transaction Service rolls back the transaction automatically. For more information, see “Transactions in EJB Applications” in *Using Transactions*.

Implementing the Process-entity Design Pattern

EJB Providers can increase application scalability by using the process-entity design pattern instead of entity beans for database access. The process-entity design pattern moves database access logic onto the server process, which achieves the following benefits:

- It reduces the server load, as the server no longer needs to manage thousands (even millions) of entity beans, each requiring transaction overhead.
- It minimizes network traffic between client applications and servers.

For more information, see the technical article *Process-Entity Design Pattern*.

Scaling Tasks for Application Assemblers and Deployers

This topic includes the following sections:

- Organizing EJBs in Groups
- Configuring the Persistent Storage Location
- Specifying the Method to Invoke Before Passivation
- Deploying Stateful Session Beans on the IIOP Listener Node
- Configuring the EJB Cache

Application Assemblers and Deployers contribute to the scalability of EJB applications by determining the optimum combinations of EJBs in an application's EJB JAR files. When partitioning EJBs, Application Assemblers and Deployers should consider the topology and resource management capabilities provided by the deployment environment. Deployers and system administrators usually collaborate on such decisions.

Organizing EJBs in Groups

When deploying EJBs, consider organizing them in particular groups:

- Wherever possible, objects that call each other should be in the same group.
- EJBs that access the same resource manager should be placed on the same group and might be packaged together in a single EJB JAR file.

Configuring the Persistent Storage Location

Stateful session beans use file-based persistent storage. For stateful session beans, the WebLogic Enterprise EJB container creates a `pstore` subdirectory in the `$APPDIR` directory to store the state information when stateful bean are passivated. When deploying WebLogic Enterprise applications, you should locate the `pstore` directory in a local file system and *not* on a NFS mounted directory.

You can change the location of the `pstore` directory by specifying the `<persistence-store-directory-root>` element in the `weblogic-ejb-extensions.xml` file, as shown in Listing 4-1.

Listing 4-1 Configuring for Persistent Storage

```
<persistence-store-descriptor>
  <persistence-store-file>
    <persistence-store-directory-root>
      /usr/me/pstore</persistence-store-directory-root>
    </persistence-store-file>
  </persistence-store-descriptor>
```

For more information about the `weblogic-ejb-extensions.xml` file, see the WebLogic Enterprise *EJB XML Reference*. You can also change this setting with the WebLogic Enterprise EJB Deployer, as described in *Using the WebLogic Enterprise EJB Deployer*.

Specifying the Method to Invoke Before Passivation

If a stateful bean is involved in a transaction, the container loads and stores the bean at appropriate times during the transaction, but does not activate or passivate the bean. If a stateful bean is not involved in a transaction, the methods `ejbLoad` and `ejbStore` are called before and after each method invocation on the bean. As stated in the Enterprise JavaBeans Specification 1.1, these load and store operations ensure that stale data is not used.

For stateful EJBs, you can specify the EJB method that is called before the EJB is stored by setting the `is-modified-method-name` element in the `weblogic-ejb-extensions.xml` file. The `is-modified-method-name` element points to the method that is called before the EJB is stored. For more information about the `weblogic-ejb-extensions.xml` file, see the WebLogic Enterprise *EJB XML Reference*. You can also change this setting with the WebLogic Enterprise EJB Deployer, as described in *Using the WebLogic Enterprise EJB Deployer*.

Deploying Stateful Session Beans on the IIOP Listener Node

Stateful session beans are conversational and therefore many messages could go to the same bean. To reduce traffic across machines, deploy stateful session beans on the node on which the IIOP Listener (ISL) runs. Clients access the WebLogic Enterprise EJB container by establishing a network connection and using the RMI on IIOP protocol to invoke EJBs. The ISL load balances incoming client connections. For more information about ISL, see “Multiplexing Incoming Client Connections” on page 1-19.

Configuring the EJB Cache

If an application encounters the `weblogic.ejb.internal.CacheFullException` exception for stateful beans, try to change the capacity of the cache or turn caching off. This exception is thrown when the WebLogic Enterprise EJB container does not find any beans that can be flushed (that is, beans in cache are either in a transaction or in a method invocation). This exception is never raised if caching is off. This exception is logged in the `ULOG` and a `RemoteException` exception, with a nested

`org.omg.CORBA.INTERNAL` exception, is returned to the client. Any client transaction involved in the request receiving the system exception `CacheFullException` will be rolled back.

EJB caching is enabled by default. Disable EJB caching for individual beans only when necessary. You can disable caching for an individual bean by setting the `is-cacheable` element in the `weblogic-ejb-extensions.xml` file. For more information about the `weblogic-ejb-extensions.xml` file, see the WebLogic Enterprise *EJB XML Reference*. You can also change this setting with the WebLogic Enterprise EJB Deployer, as described in *Using the WebLogic Enterprise EJB Deployer*.

Scaling Tasks for System Administrators

System administrators contribute to the scalability of EJB applications by configuring and tuning the deployment environment for optimum application performance. System Administrators can increase application performance by:

- Replicating servers and server groups, as described in “Replicating Server Processes and Server Groups” on page 1-9.
- Using multithreaded Java servers, as described in “Using Multithreaded Java Servers (Java only)” on page 1-13. In general, EJB applications perform better when running on multithreaded Java servers.
- Supporting additional incoming client connections, as described in “Multiplexing Incoming Client Connections” on page 1-19.
- Removing Orphan Files for File-based Persistence, as described later in this section.
- Scaling and Tuning the EJB Cache, as described later in this section.

Removing Orphan Files for File-based Persistence

System administrators should periodically remove orphan files associated with stateful session beans that use file-based persistence.

With file-based persistence, WebLogic Enterprise stores the bean's state in a file in a directory, which is either the `pstore` subdirectory in the `$APPDIR` directory, or the directory specified by the setting of the `persistence-store-directory-root` XML element in the `weblogic-ejb-extensions.xml` file. If the client process crashes (for example, the network connection is lost or the client machine is turned off), the file is not automatically removed. An accumulation of these files can slow performance.

System Administrators can create startup scripts that delete these files whenever the WebLogic Enterprise environment is shut down and restarted. For more information, see “Starting and Shutting Down Applications” in the *Administration Guide*. For more information about monitoring and tuning the performance of a WebLogic Enterprise system, see “Monitoring a Running System” in the *Administration Guide*. For more information about the `weblogic-ejb-extensions.xml` file, see the WebLogic Enterprise *EJB XML Reference*.

Scaling and Tuning the EJB Cache

System administrators can scale and tune EJB applications by using the WebLogic Enterprise caching features for entity beans and stateful session beans. The WebLogic Enterprise EJB container supports caching beans across method invocations as well as across transactions. This capability improves performance by significantly reducing the frequency of beans being passivated. EJB caching is enabled by default for stateful beans.

A cached stateful bean is normally stored (passivated) only if it is unused for a period of time. You can configure an optional cache flush time, if desired. A bean may be considered unused if other beans are being used more frequently and the bean is the least recently used bean. The bean may also be passivated if the cache flush time occurs and the bean is not presently active within a method call or a transaction.

Mechanisms For Managing the WebLogic Enterprise EJB Cache

EJB caching is enabled by default for stateful session beans and entity beans in the EJB container. You can set up caching using the following mechanisms:

- Via the WebLogic EJB extensions to the deployment descriptor DTD. You can use the `is-cacheable` element to disable caching for individual beans. For more information, see “Configuring the EJB Cache” on page 4-7.

- Via JavaServer parameters in the `UBBCONFIG` file. You can configure the following two settings for bean caching:
 - `MAXEJBCACHE` is the maximum number of beans that can be cached at any one time.
 - `EJBCACHE_FLUSH` is the number of minutes between cache flushes, when the bean cache is to be flushed by the system. You can specify the number of minutes between cache flushes. At the interval specified by the cache flush time, all beans that are not currently in a transaction or a method invocation are passivated and their memory is freed.

For more information about these parameters, see “Creating a Configuration File” in the *Administration Guide*.

Tuning the Cache

Sizing the bean cache correctly is very important. If you are using multithreaded servers, the cache should at least be the number of threads in the server. If the cache is smaller than the number of threads, applications could encounter the `CacheFullException` exception because all the beans (one per thread) are active in a method invocation.

The optimum number of beans to maintain in the cache should be based on how many beans can be active simultaneously in the server process. This number is determined by:

- How long a bean will remain active in a server before it can be removed or is dormant.
- How many threads the server has.

Note the following regarding tuning and scaling the bean cache:

- While a bean is cached in a server process, all requests for the bean return to the server process that has cached the bean. Caching a bean effectively disables load balancing for the bean. The advantage of caching a bean is that caching saves a lot of activation and passivation, which involves persistent storage I/O and uses Java File Serialization to store conversational state.

Every bean is passivated immediately after creation to give an opportunity to balance the load. This approach incurs at least two I/O cycles during the lifecycle of a stateful bean, even though the lifecycle may be relatively short.

This could change in the future based on customer input regarding the pattern in which stateful beans are used.

In general, BEA recommends not flushing the cache frequently. However, after a cache flush, all the beans not then active (in a method or a transaction) are again available for load balancing to servers supporting the bean.

- If a server process in which a cached bean exists crashes, there is no recovery of that bean's state.
- Every bean cache entry potentially uses an Active Object Map (AOM) entry. The default 1,000 objects in the AOM parameter specified in the `UBBCONFIG` file may be insufficient if you have many server processes with many large caches.

5 Distributing Applications

This topic includes the following sections:

- Why Distribute an Application?
- Using Data-dependent Routing (BEA Tuxedo Servers Only)
- Configuring the UBBCONFIG File
- Configuring the factory_finder.ini (CORBA Applications Only)
- Modifying the Domain Gateway Configuration File to Support Routing

This topic describes how to distribute applications in the WebLogic Enterprise environment, using a CORBA application as an example. However, the concepts apply to EJB applications as well. For more information about EJB applications, see “Scaling Tasks for Application Assemblers and Deployers” on page 4-5.

Why Distribute an Application?

This topic includes the following sections:

- About Distributing an Application
- Benefits of a Distributed Application
- Characteristics of Distributing an Application

About Distributing an Application

Distributing an application enables you to select which parts of an application should be grouped together logically and where these groups should run. You distribute an application by creating more than one entry in the `GROUPS` section of the `UBBCONFIG` file, and by dividing application resources or tasks among the groups. Creating groups of servers enables you to partition a very large application into its component business applications, and to assure that each of these into logical components is of a manageable size and in an optimal location.

Benefits of a Distributed Application

The benefits of a distributed application include:

- **Scalability.** To increase the load that an application can sustain:
 - Place extra server processes in a group.
 - Add machines to the application and redistribute the groups across the machines.
 - Replicate a group onto other machines within the application and use load balancing.
 - Segment a database and use data-dependent routing to reach the groups dealing with these separate database segments (the BEA Tuxedo system).

With the WebLogic Enterprise system, you can use factory-based routing to distribute the processing of a particular CORBA interface across multiple server groups and, if desired, across multiple machines. This feature allows you to distribute the processing load, which can prevent the processing bottlenecks that occur when concurrent, resource-intensive applications compete for the available CPU, memory, disk I/O, and network resources. For an example of using factory-based routing, see “Scaling with Factory-based Routing” on page 2-11.

For more information about WebLogic Enterprise scalability features, see Chapter 1, “Scaling WebLogic Enterprise Applications.”

- **Ease of Development and Maintenance.** The separation of the business application logic into services or components that communicate through well-defined messages or interfaces allows both development and maintenance to be similarly separated and thereby simplified.
- **Reliability.** When multiple machines are in use and one fails, the remainder can continue operation. Similarly, when multiple server processes are within a group and one fails, the others are available to perform work. Finally, if a machine should fail, but there are multiple machines within the application, these other machines can be used to handle the load.
- **Coordination of Autonomous Actions.** If you have separate applications, you can coordinate autonomous actions, as a single logical *unit of work*, among applications. *Autonomous actions* are actions that involve multiple server groups and multiple resource manager interfaces.

Characteristics of Distributing an Application

A distributed application:

- Enlarges the client and/or server model.
- Establishes multiple server groups.
- Enables transparent access to BEA Tuxedo services or WebLogic Enterprise interfaces.
- In BEA Tuxedo, allows data-dependent partitioning of data.

- In WebLogic Enterprise, allows partitioning of CORBA objects in multiple groups across multiple machines, or distributing application factory interfaces and application interfaces.
- Enables management of multiple resources.
- Supports a networked model.

Using Data-dependent Routing (BEA Tuxedo Servers Only)

This topic includes the following sections:

- About Data-dependent Routing
- Characteristics of Data-dependent Routing
- Sample Distributed Application

Note: This topic applies to BEA Tuxedo servers only.

About Data-dependent Routing

Data-dependent routing is a mechanism whereby a service request is routed by a client (or a server acting as a client) to a server within a specific group based on a data value contained within the buffer that is sent. Within the internal code of a service call, BEA Tuxedo chooses a destination server by comparing a data field with the routing criteria it finds in the Bulletin Board shared memory.

For any given service, a routing criteria identifier can be specified in the `SERVICES` section of the `UBBCONFIG` file. The routing criteria identifier (in particular, the mapping of data ranges to server groups) is specified in the `ROUTING` section.

Characteristics of Data-dependent Routing

Data-dependent routing has the following characteristics:

- The service request assigned to a server in the group is based on a data value.
- Routing uses the Bulletin Board criteria and occurs in a server call.
- The routing criteria identifier for a service is specified in the `SERVICES` section of the `UBBCONFIG` file.
- The routing criteria identifier is defined in the `ROUTING` section of the `UBBCONFIG` file.

Sample Distributed Application

Table 5-1 illustrates how client requests are routed to servers. In this example, a banking application called `bankapp` uses data-dependent routing. For `bankapp`, there are three groups (`BANKB1`, `BANKB2`, and `BANKB3`), and two routing criteria (`Account_ID` and `Branch_ID`). The services `WITHDRAW`, `DEPOSIT`, and `INQUIRY` are routed using the `Account_ID` field. The services `OPEN` and `CLOSE` are routed using the `Branch_ID` field.

Table 5-1 Data Dependent Routing Criteria for Sample Distributed Application

Server Group	Routing Criteria	Services
BANKB1	Account_ID: 10000 - 49999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 1 - 4	OPEN and CLOSE
BANKB2	Account_ID: 50000 - 79999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 5 - 7	OPEN and CLOSE
BANKB3	Account_ID: 80000 - 109999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 8 - 10	OPEN and CLOSE

Configuring the UBBCONFIG File

This topic includes the following sections:

- About the UBBCONFIG File in Distributed Applications
- Modifying the GROUPS Section
- Modifying the SERVICES Section
- Creating the ROUTING Section
- Example of UBBCONFIG Sections in a Distributed Application

For more information about the UBBCONFIG file, see “Creating a Configuration File” in the *Administration Guide*.

About the UBBCONFIG File in Distributed Applications

The UBBCONFIG file contains a description of either data-dependent routing (BEA Tuxedo) or factory-based routing (WebLogic Enterprise CORBA), as follows:

- The GROUPS section is populated with as many server groups as are required for distributing the system. This allows the system to route a request to a server in a specific group. These groups can all reside on the same site (SHM mode) or, if there is networking, the groups can reside on different sites (MP mode).
- For data-dependent routing in BEA Tuxedo, the SERVICES section must list the routing criteria for each service that uses the ROUTING parameter.
Note: If a service has multiple entries, each with a different SRVGRP parameter, all such entries must set ROUTING the same way to ensure consistency for that service. A service can route only on one field, which must be the same for all the same services.
- For factory-based routing in WebLogic Enterprise, the INTERFACES section must list the name of the routing criteria for each CORBA interface that uses the FACTORYROUTING parameter. This parameter is set to the name of a routing criteria defined in the ROUTING section.

- Add a `ROUTING` section to the configuration file to show mappings between data ranges and groups so that the system can send the request to a server in a specific group. Each `ROUTING` section item contains an identifier that is used in the `INTERFACES` section (for WebLogic Enterprise) or in the `SERVICES` section (for BEA Tuxedo).

Modifying the GROUPS Section

The parameters in the `GROUPS` section implement two important aspects of distributed transaction processing:

- They associate a group of servers with a particular `LMID` and a particular instance of a resource manager.
- By allowing a second `LMID` to be associated with the server group, they name an alternate machine to which a group of servers can be migrated if the `MIGRATE` option is specified.

Table 5-2 describes the parameters in the `GROUPS` section.

Table 5-2 Parameters Specified in the GROUPS Section

Parameter	Meaning
LMID	LMID must be assigned in the <code>MACHINES</code> section to indicate that this server group runs on this particular machine. A second <code>LMID</code> value can be specified (separated from the first by a comma) to name an alternate machine to which this server group can be migrated if the <code>MIGRATE</code> option has been specified. Servers in the group must specify <code>RESTART=Y</code> to migrate.
GRPNO	Associates a numeric group number with this server group. The number must be greater than zero (0) and less than 30000. It must be unique among entries in the <code>GROUPS</code> section in this configuration file. (Required)
TMSNAME	Specifies which transaction management server (TMS) should be associated with this server group.

Table 5-2 Parameters Specified in the GROUPS Section (Continued)

Parameter	Meaning
TMSCOUNT	Specifies how many copies of TMSNAME should be started for this server group. The minimum value is 2. If not specified, the default is 3. All TMSNAME servers started for a server group are automatically set up in an MSSQ set. (Optional)
OPENINFO	<p>Specifies information needed to open a particular instance of a particular resource manager, or it indicates that such information is not required for this server group. When a resource manager is named in the OPENINFO parameter, information such as the name of the database and the access mode is included. The entire value string must be enclosed in double quotes and must not be more than 256 characters. The format of the OPENINFO string is dependent on the requirements of the vendor providing the underlying resource manager. The string required by the vendor must be prefixed with <code>rm_name:</code>, which is the published name of the vendor's transaction (XA) interface followed immediately by a colon (<code>:</code>).</p> <p>The OPENINFO parameter is ignored if TMSNAME is not set or is set to TMS. If TMSNAME is set but the OPENINFO string is set to the null string (<code>" "</code>) or if this parameter does not appear on the entry, it means that a resource manager exists for the group but does not require any information for executing an open operation.</p>
CLOSEINFO	Specifies information the resource manager needs when closing a database. The parameter can be omitted or the null string can be specified. The default is the null string.

Modifying the SERVICES Section

The SERVICES section contains parameters that control the way application services are handled. An entry line in this section is associated with a service by its identifier name. Because the same service can be link edited with more than one server, the SRVGRP parameter is provided to tie the parameters for an instance of a service to a particular group of servers.

Parameters to Modify

Three parameters in the `SERVICES` section are particularly related to DTP: `ROUTING`, `AUTOTRAN`, and `TRANTIME`.

Table 5-3 describes the parameters in the `SERVICES` section.

Table 5-3 Parameters Specified in the `SERVICES` Section

Parameter	Meaning
<code>ROUTING</code>	Points to an entry in the <code>ROUTING</code> section where data-dependent routing is specified for transactions that request this service.
<code>AUTOTRAN</code>	Determines whether a transaction should be started automatically if a message received by this service is not already in transaction mode. The default is <code>N</code> . Use of the parameter should be coordinated with the programmers that code the services for your application. CORBA applications only.
<code>TRANTIME</code>	Specifies a timeout value, in seconds, for transactions automatically started in this service. The default is 30 seconds. Required only if <code>AUTOTRAN=Y</code> and another timeout value is needed. CORBA and RMI applications only.

Note: `AUTOTRAN` and `TRANTIME` apply to CORBA and RMI applications only. For EJB applications, the `AUTOTRAN` parameter is ignored and the transaction timeout is specified in the `<trans-timeout-seconds>` XML element in the `weblogic-ejb-extensions.xml` file. For more information, see “Transactions in EJB Applications” in *Using Transactions*.

Sample `SERVICES` Section

Listing 5-1 shows a sample `SERVICES` section.

Listing 5-1 Sample `SERVICES` Section

```
*SERVICES

WITHDRAW  ROUTING=ACCOUNT_ID
```

```
DEPOSIT    ROUTING=ACCOUNT_ID
OPEN_ACCT  ROUTING=BRANCH_ID
```

Creating the ROUTING Section

For information about ROUTING parameters that support BEA Tuxedo data-dependent routing or the WebLogic Enterprise factory-based routing, see “Creating a Configuration File” in the *Administration Guide*.

Example of UBBCONFIG Sections in a Distributed Application

Listing 5-2 shows a sample UBBCONFIG file that contains the GROUPS, SERVICES, and ROUTING sections of a configuration file to accomplish data-dependent routing in the BEA Tuxedo system.

Listing 5-2 Sample UBBCONFIG File

```
*GROUPS
BANKB1          GRPNO=1
BANKB2          GRPNO=2
BANKB3          GRPNO=3
#
*SERVICES
WITHDRAW        ROUTING=ACCOUNT_ID
DEPOSIT         ROUTING=ACCOUNT_ID
INQUIRY         ROUTING=ACCOUNT_ID
OPEN_ACCT       ROUTING=BRANCH_ID
CLOSE_ACCT      ROUTING=BRANCH_ID
#
*ROUTING
ACCOUNT_ID      FIELD=ACCOUNT_ID BUFTYPE="FML"
                RANGES="MIN - 9999:*,
                10000-49999:BANKB1,
                50000-79999:BANKB2,
                80000-109999:BANKB3,
                *: *"
```



```
BRANCH_ID          FIELD=BRANCH_ID BUFTYPE="FML"
                   RANGES="MIN - 0:* ,
                           1-4:BANKB1 ,
                           5-7:BANKB2 ,
                           8-10:BANKB3 ,
                           * : * "
```

Configuring the factory_finder.ini (CORBA Applications Only)

For CORBA applications, to configure factory-based routing across multiple domains, you must configure the `factory_finder.ini` file to identify factory objects that are used in the current (local) domain but that are resident in a different (remote) domain. For more information, see “Configuring Multiple Domains (WebLogic Enterprise System)” in the *Administration Guide*.

Modifying the Domain Gateway Configuration File to Support Routing

This topic includes the following sections:

- About the Domain Gateway Configuration File
- Parameters in the DM_ROUTING Section of the DMCONFIG File (BEA Tuxedo Only)

This section is specific to BEA Tuxedo and explains how and why you need to modify the domain gateway configuration to support routing. For more information about the domain gateway configuration file, see “Configuring Multiple Domains (WebLogic Enterprise System)” in the *Administration Guide*.

About the Domain Gateway Configuration File

The Domain gateway configuration information is stored in a binary file called `BDMCONFIG`. The `DMCONFIG` file (ASCII) is created and edited with any text editor. The compiled `BDMCONFIG` file can be updated while the system is running by using the `dmadmin(1)` command.

You must have one `BDMCONFIG` file for each BEA Tuxedo application that requires the Domains functionality. System access to the `BDMCONFIG` file is provided through the Domains administrative server, `DMADM(5)`. When a gateway group is booted, the gateway administrative server, `GWADM(5)`, requests from the `DMADM` server a copy of the configuration required by that group. The `GWADM` server and the `DMADM` server also ensure that run-time changes to the configuration are reflected in the corresponding Domain gateway groups.

Note: For more information about modifying the `DMCONFIG` file, see “Configuring Multiple Domains (WebLogic Enterprise System)” in the *Administration Guide*.

Parameters in the `DM_ROUTING` Section of the `DMCONFIG` File (BEA Tuxedo Only)

The `DM_ROUTING` section provides information for data-dependent routing of service requests using `FML`, `VIEW`, `X_C_TYPE`, and `X_COMMON` typed buffers. Lines within the `DM_ROUTING` section have the form `CRITERION_NAME`, where `CRITERION_NAME` is the (identifier) name of the routing entry specified in the `SERVICES` section. The `CRITERION_NAME` entry may contain no more than 15 characters.

Parameters to Specify

Table 5-4 describes the parameters in the `DM_ROUTING` section.

Table 5-4 Parameters Specified in the DM_ROUTING Section

Parameter	Description
<i>FIELD = identifier</i>	Specifies the name of the routing field. It must contain 30 characters or fewer. This field is assumed to be a field name identified in an FML field table (for FML buffers) or an FML VIEW table (for VIEW, X_C_TYPE, or X_COMMON buffers). The FLDTBLDIR and FIELDTBLS environment variables are used to locate FML field tables; the VIEWDIR and VIEWFILES environment variables are used to locate FML VIEW tables. If a field in an FML32 buffer is used for routing, it must have a field number less than or equal to 8191.
BUFTYPE = "type1[:subtype1[,subtype2 . . .]][:type2[:subtyp e3[. . .]]] . . ."	<p>Specifies list of types and subtypes of data buffers for which this routing entry is valid. The types are restricted to FML, VIEW, X_C_TYPE, and X_COMMON.</p> <p>No subtype can be specified for type FML, and subtypes are required for the other types (* is not allowed).</p> <p>Duplicate type/subtype pairs cannot be specified for the same routing criteria name; more than one routing entry can have the same criteria name as long as the type/subtype pairs are unique. This parameter is required.</p> <p>If multiple buffer types are specified for a single routing entry, the data types of the routing field for each buffer type must be the same. (If the field value is not set (for FML buffers), or does not match any specific range, and a wildcard range has not been specified, then an error is returned to the application process that requested the execution of the remote service.)</p>

Table 5-4 Parameters Specified in the DM_ROUTING Section (Continued)

Parameter	Description
<i>RANGES</i> ="range1:rdom1[,range2:rdom2 ...]"	<p>Specifies the ranges and associated remote domain names (RDOM) for the routing field. The string must be enclosed in double quotes, with the format of a comma-separated ordered list of range/RDOM pairs.</p> <p>A range is either a single value (signed numeric value or character string in single quotes), or a range of the form <i>lower - upper</i> (where lower and upper are both signed numeric values or character strings in single quotes). The value of <i>lower</i> must be less than or equal to <i>upper</i>. A single quote embedded in a character string value (such as "O'Brien"), must be preceded by two backslashes ("O\\'Brien").</p> <ul style="list-style-type: none">■ Use MIN to indicate the minimum value for the data type of the associated FIELD. For strings and carrays, it is the null string; for character fields, it is 0; for numeric values, it is the minimum numeric value that can be stored in the field.■ Use MAX to indicate the maximum value for the data type of the associated FIELD. For strings and carrays, it is effectively an unlimited string of octal-255 characters; for a character field, it is a single octal-255 character; for numeric values, it is the maximum numeric value that can be stored in the field. <p>Thus, MIN - -5 is all numbers less than or equal to -5, and 6 - MAX is all numbers greater than or equal to 6.</p> <p>The metacharacter * (wildcard) in the position of a range indicates any values not covered by the other ranges previously seen in the entry. Only one wildcard range is allowed per entry and it should be last (ranges following it are ignored).</p>

Routing Field Description

The routing field can be of any data type supported in FML or VIEW. A numeric routing field must have numeric range values, and a string routing field must have string range values.

String range values for string, carray, and character field types must be placed inside a pair of single quotation marks and cannot be preceded by a sign. Short and long integer values are a string of digits, optionally preceded by a plus (+) or minus (-) sign.

Floating point numbers are of the form accepted by the C compiler or `atof()`: an optional sign, followed by a string of digits optionally containing a decimal point, and an optional `e` or `E` followed by an optional sign or space, and an integer.

When a field value matches a range, the associated `RDOM` value specifies the remote domain to which the request should be routed. An `RDOM` value of `*` indicates that the request can go to any remote domain known by the gateway group. Within a `range/RDOM` pair, the range is separated from the `RDOM` by a `:` (colon).

Example of a Five-Site Domain Configuration Using Routing

Listing 5-3 shows a configuration file that defines a five-site domain configuration. It has four bank branch domains communicating with a Central Bank Branch. Three of the bank branches run within other BEA Tuxedo system domains. The fourth branch runs under the control of another TP domain, and OSI-TP is used in the communication with that domain. The example shows the BEA Tuxedo Domain gateway configuration file from the Central Bank point of view. In the `DM_TDOMAIN` section, this example shows a mirrored gateway for `b01`.

Listing 5-3 DMCONFIG File for a Five-Site Domains Configuration

```
# BEA TUXEDO DOMAIN CONFIGURATION FILE FOR THE CENTRAL BANK
#
#
*DM_LOCAL_DOMAINS
# <local domain name> <Gateway Group name> <domain type> <domain id> <log device>
#                               [<audit log>] [<blocktime>]
#                               [<log name>] [<log offset>] [<log size>]
#                               [<maxrdom>] [<maxrdtran>] [<maxtran>]
#                               [<maxdatalen>] [<security>]
#                               [<tuxconfig>] [<tuxoffset>]
#
#
DEFAULT: SECURITY = NONE
c01      GWGRP = bankg1
         TYPE = TDOMAIN
         DOMAINID = "BA.CENTRAL01"
         DMTLOGDEV = "/usr/apps/bank/DMTLOG"
         DMTLOGNAME = "DMTLG_C01"
c02      GWGRP = bankg2
         TYPE = OSITP
         DOMAINID = "BA.CENTRAL01"
```

5 *Distributing Applications*

```
DMTLOGDEV = "/usr/apps/bank/DMTLOG"
DMTLOGNAME = "DMTLG_C02"
NWDEVICE = "OSITP"
URCH = "ABCD"

#
*DM_REMOTE_DOMAINS
#<remote domain name> <domain type> <domain id>
#
b01      TYPE = TDOMAIN
          DOMAINID = "BA.BANK01"
b02      TYPE = TDOMAIN
          DOMAINID = "BA.BANK02"
b03      TYPE = TDOMAIN
          DOMAINID = "BA.BANK03"
b04      TYPE = OSITP
          DOMAINID = "BA.BANK04"
          URCH = "ABCD"

#
*DM_TDOMAIN
#
#      <local or remote domainname> <network address> [nwdevice]
#
# Local network addresses
c01      NWADDR = "//newyork.acme.com:65432"      NWDEVICE = "/dev/tcp"
c02      NWADDR = "//192.76.7.47:65433"      NWDEVICE = "/dev/tcp"
# Remote network addresses: second b01 specifies a mirrored gateway
b01      NWADDR = "//192.11.109.5:1025" NWDEVICE = "/dev/tcp"
b01      NWADDR = "//194.12.110.5:1025" NWDEVICE = "/dev/tcp"
b02      NWADDR = "//dallas.acme.com:65432" NWDEVICE = "/dev/tcp"
b03      NWADDR = "//192.11.109.156:4244" NWDEVICE = "/dev/tcp"
#
*DM_OSITP
#
#<local or remote domain name> <apt> <aeq>
#                                [<aet>] [<acn>] [<apid>] [<aeid>]
#                                [<profile>]
#
c02      APT = "BA.CENTRAL01"
          AEQ = "TUXEDO.R.4.2.1"
          AET = "{1.3.15.0.3},{1}"
          ACN = "XATMI"
b04      APT = "BA.BANK04"
          AEQ = "TUXEDO.R.4.2.1"
          AET = "{1.3.15.0.4},{1}"
          ACN = "XATMI"
*DM_LOCAL_SERVICES
#<service_name>  [<Local Domain name>] [<access control>] [<exported svcname>]
#                [<inbuftype>] [<outbuftype>]
#
```

Modifying the Domain Gateway Configuration File to Support Routing

```
open_act      ACL = branch
close_act     ACL = branch
credit
debit
balance
loan          LDOM = c02      ACL = loans
*DM_REMOTE_SERVICES
#<service_name>  [<Remote domain name>] [<local domain name>]
#               [<remote svcname>] [<routing>] [<conv>]
#               [<trantime>] [<inbuftype>] [<outbuftype>]
#
tlr_add       LDOM = c01  ROUTING = ACCOUNT
tlr_bal       LDOM = c01  ROUTING = ACCOUNT
tlr_add       RDOM = b04  LDOM = c02  RNAME = "TPSU002"
tlr_bal       RDOM = b04  LDOM = c02  RNAME = "TPSU003"
*DM_ROUTING
# <routing criteria>    <field> <typed buffer> <ranges>
#
ACCOUNT FIELD = branchid  BUFTYPE = "VIEW:account"
          RANGES = "MIN - 1000:b01, 1001-3000:b02, *:b03"
*DM_ACCESS_CONTROL
#<acl name>    <Remote domain list>
#
branch  ACLIST = b01, b02, b03
loans   ACLIST = b04
```

6 Tuning Applications

This topic includes the following sections:

- Maximizing Application Resources
- When to Use MSSQ Sets (BEA Tuxedo Servers Only)
- Enabling Load Balancing
- Configuring Replicated Server Processes and Groups
- Configuring Multithreaded Java Servers
- Assigning Priorities to Interfaces or Services
- Bundling Services into Servers (BEA Tuxedo Servers Only)
- Enhancing Efficiency with Application Parameters
- Setting Application Parameters
- Determining IPC Requirements
- Measuring System Traffic

For more information about monitoring WebLogic Enterprise applications, see “Monitoring a Running System” in the *Administration Guide*.

Maximizing Application Resources

Making correct decisions in the following areas can improve the functioning of your WebLogic Enterprise or BEA Tuxedo applications:

- When to use MSSQ sets (BEA Tuxedo).
- How to assign load factors.
- How to package interfaces and/or services into servers.
- How to set application parameters.
- How to tune operating system IPC parameters.
- How to detect and eliminate bottlenecks.

When to Use MSSQ Sets (BEA Tuxedo Servers Only)

Note: MSSQ sets are not supported in WebLogic Enterprise.

Table 6-1 describes when to use MSSQ sets with BEA Tuxedo servers.

Table 6-1 When and When Not to Use MSSQ Sets

Use MSSQ Sets When	Do Not Use MSSQ Sets When
There are several, but not too many servers.	There is a large number of servers. (A compromise is to use many MSSQ sets.)
Buffer sizes are not too large.	Buffer sizes are large enough to exhaust one queue.
The servers offer identical sets of services.	Services are different for each server.

Table 6-1 When and When Not to Use MSSQ Sets (Continued)

Use MSSQ Sets When	Do Not Use MSSQ Sets When
The messages involved are reasonably sized.	Long messages are being passed to the services causing the queue to be exhausted. This causes nonblocking sends to fail, or blocking sends to block.
Optimization and consistency of service turnaround time is paramount.	Optimization and consistency of service turnaround time is not critical.

The following two analogies help to show why using MSSQ sets is sometimes, but not always, beneficial:

- An application in which MSSQ sets are used appropriately is similar to a bank, where all the tellers offer the same services and customers wait in line for the first available teller. This efficient arrangement ensures the best use of available services.
- An application in which it is better to avoid using MSSQ sets is similar to a supermarket, where each cashier offers a different set of services: some accept cash only, some accept credit cards, and still others serve only customers buying fewer than ten items.

Enabling Load Balancing

This topic includes the following sections:

- About Load Balancing
- Two Ways to Measure Service Performance Time (BEA Tuxedo Servers Only)

About Load Balancing

On BEA Tuxedo systems, you can control whether a load balancing algorithm is used on the system as a whole. With load balancing, a load factor is applied to each service within the system, and you can track the total load on every server. Every service request is sent to the qualified server that is least loaded.

Note: On WebLogic Enterprise systems, load balancing is enabled automatically. You *cannot* disable load balancing by specifying `LDBAL=N`.

To determine how to assign load factors (located in the `SERVICES` section), run an application continually and calculate the average time it takes for each service to be performed. Assign a `LOAD` value of 50 (`LOAD=50`) to any service that requires the average amount of time that you calculated. Any service taking longer to execute than the calculated average should have a `LOAD>50`. Any service taking less to execute than the calculated average should have a `LOAD<50`.

A `LOAD` factor is assigned to each service performed, which keeps track of the total load of services that each server has performed. Each service request is routed to the server with the smallest total load. The routing of that request causes the server's total to be increased by the `LOAD` factor of the service requested.

You can also apply `LOAD` factors to interfaces. For more information about `LOAD` factors, see “Creating a Configuration File” in the *Administration Guide*.

Two Ways to Measure Service Performance Time (BEA Tuxedo Servers Only)

You can measure service performance time in one of the following ways:

- Enter `servopts -r` in the configuration file. The `-r` option causes a log of services performed to be written to standard error. You can then use the `txrpt(1)` command to analyze this information. For details about `servopts(5)` and `txrpt(1)`, see “Section 1 - Commands” in the *BEA Tuxedo Reference Manual*.
- Insert calls to `time(2)` at the beginning and end of a service routine. Services that take the longest time receive the highest load. Those that take the shortest time receive the lowest load. For details about `time(2)`, see a UNIX system reference manual.

Configuring Replicated Server Processes and Groups

To configure replicated server processes and groups in the WebLogic Enterprise domain, complete the following steps:

1. Edit the application’s `UBBCONFIG` file using a text editor.
2. In the `GROUPS` section, specify the names of the groups you want to configure.

3. In the `SERVERS` section, specify the parameters in Table 6-2 for the server process you want to replicate.

Table 6-2 Parameters Specified in the `SERVERS` Section

Parameter	Description
Server application name	<ul style="list-style-type: none"> ■ For Java, this is the name of the executable file for the Java server, plus the name of the JAR file that will be dynamically loaded with the server boots. ■ For C++, this is the name of the executable file that contains the application server.
GROUP	Specifies the name of the group to which the server process belongs. If you are replicating a server process across multiple groups, specify the server process once for each group.
SRVID	Specifies a numeric identifier, giving the server process a unique identity.
MIN	Specifies the number of instances of the server process to start when you start the application.
MAX	Specifies the maximum number of server processes that can be running at any one time.

The `MIN` and `MAX` parameters determine the degree to which a given server application can process requests on a given interface in parallel. During run time, the system administrator can examine resource bottlenecks and start additional server processes, if necessary, thereby scaling the application. For more information, see “Monitoring a Running Application” in the *Administration Guide*.

Note: The `MAX` parameter controls the maximum number of instances. However, WebLogic Enterprise does not spawn instances automatically. The system will automatically start up to the specified `MIN` number of instances. Between `MIN` and `MAX`, the system administrator will need to spawn new instances manually. Once `MAX` is reached, an error will be returned by `tmboot`, `tmadmin`, or the `TMIB API`.

Configuring Multithreaded Java Servers

This topic includes the following sections:

- Setting the OPENINFO Parameter
- Configuring the Number of Threads
- Configuring the Number of Concurrent Accessors

For more information about multithreaded Java servers, see “Using Multithreaded Java Servers (Java only)” on page 1-13.

Setting the OPENINFO Parameter

To configure a multithreaded Java server, you must add `Threads=true` to the `OPENINFO` parameter in the `GROUPS` section of the `UBBCONFIG` file, as shown in Listing 6-1.

Listing 6-1 Adding `Threads=true` to the `OPENINFO` Parameter

```
OPENINFO="ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=
.+MaxCur=5+Threads=true"
```

Configuring the Number of Threads

You can establish the number of threads for a Java server application by using the `-M` option to the `JavaServer` parameter. This parameter is used in the `SERVERS` section of the application’s `UBBCONFIG` file. For a description of the `-M` options, see “Creating a Configuration File” in the *Administration Guide*.

For multithreaded WebLogic Enterprise Java servers, you must account for the number of worker threads that each server is configured to run. Worker threads are threads that are started and managed by the WebLogic Enterprise Java software, as opposed to

threads started and managed by an application program. Internally, WebLogic Enterprise Java manages a pool of available worker threads. When a client request is received, an available worker thread from the thread pool is scheduled to execute the request. There is one thread per active object, and while the object is active, the thread is busy. When the request is done, the worker thread is returned to the pool of available threads.

Configuring the Number of Concurrent Accessors

The `MAXACCESSERS` parameter in the application's `UBBCONFIG` file sets the maximum number of concurrent accessors of a WebLogic Enterprise system. Accessors include native and remote clients, servers, and administration processes.

A single-threaded server counts as one accessor. For a multithreaded Java server, the number of accessors can be up to twice the maximum number of worker threads that the server is configured to run, plus one for the server itself. However, to calculate a `MAXACCESSERS` value for a WebLogic Enterprise system running multithreaded servers, *do not* simply double the existing `MAXACCESSERS` value of the whole system. Instead, you add up the accessors for each multithreaded server.

For example, assume that you have three multithreaded Java servers in your system:

- Java server A is configured to run three worker threads.
- Java server B is configured to run four worker threads.
- Java server C is configured to run five worker threads.

The accessor requirement of these servers is calculated by using the following formula:

$$[(3 * 2) + 1] + [(4 * 2) + 1] + [(5 * 2) + 1] = 27 \text{ accessors}$$

Assigning Priorities to Interfaces or Services

This topic includes the following sections:

- About Priorities to Interfaces or Services
- Characteristics of the PRIO Parameter

About Priorities to Interfaces or Services

You can exert significant control over the flow of data in an application by assigning priorities to BEA Tuxedo services using the `PRIO` parameter. For an application running on a BEA Tuxedo system, you can specify the `PRIO` parameter for each service named in the `SERVICES` section of the application's `UBBCONFIG` file.

For example, Server 1 offers Interfaces A, B, and C. Interfaces A and B have a priority of 50 and Interface C has a priority of 70. An interface requested for C is always dequeued before a request for A or B. Requests for A and B are dequeued equally with respect to one another. The system dequeues every tenth request in first-in, first-out (FIFO) order to prevent a message from waiting indefinitely on the queue.

For Tuxedo and native C++ CORBA applications (but not Java applications), you can also dynamically change a priority with the `tpsprio()` call. Only preferred clients should be able to increase the service priority. In a system on which servers perform service requests, the server can call `tpsprio()` to increase the priority of its interface or service calls so the user does not wait in line for every interface or service request that is required.

Characteristics of the PRIO Parameter

The `PRIO` parameter should be used carefully. Depending on the order of messages on the queue (for example, A, B, and C), some (such as A and B) will be dequeued only one in ten times. This means reduced performance and potential slow turnaround time on the service.

The characteristics of the `PRIO` parameter are as follows:

- It determines the priority of an interface or a service on the server's queue.
- The highest assigned priority gets first preference. This interface or service should occur less frequently.
- A lower priority message does not remain forever enqueued, because every tenth message is retrieved on a FIFO basis. Response time should not be a concern of the lower priority interface or service.

Assigning priorities enables you to provide more efficient service to the most important requests and slower service to the less important requests. You can also give priority to specific users or in specific circumstances.

Bundling Services into Servers (BEA Tuxedo Servers Only)

This topic includes the following sections:

- About Bundling Services
- When to Bundle Services

About Bundling Services

The easiest way to package services into server executables is to not package them at all. Unfortunately, if you do not package services, the number of server executables, and also message queues and semaphores, rises beyond an acceptable level. There is a trade-off between not bundling services and bundling services too much.

When to Bundle Services

You should bundle services for the following reasons:

- **Functional similarity.** If some services are similar in their role in the application, you can bundle them in the same server. The application can offer all or none of them at a given time. An example is the `bankapp` application, in which the `WITHDRAW`, `DEPOSIT`, and `INQUIRY` services are all teller operations. Administration of services becomes simpler.
- **Similar libraries.** For example, if you have three services that use the same 100K library and three services that use different 100K libraries, bundling the first three services saves 200K. Often, functionally equivalent services have similar libraries.
- **Filling the queue.** Bundle only as many services into a server as the queue can handle. Each service added to an unfilled MSSQ set may add relatively little to the size of an executable, and nothing to the number of queues in the system. Once the queue is filled, however, the system performance degrades and you must create more executables to compensate.
- **Placement of call-dependent services.** Avoid placing, in the same server, two (or more) services that call each other. If you do so, the server will issue a call to itself, causing a deadlock.

Enhancing Efficiency with Application Parameters

This topic includes the following sections:

- Setting the `MAXACCESSERS`, `MAXSERVERS`, `MAXINTERFACES`, and `MAXSERVICES` Parameters
- Setting the `MAXGTT`, `MAXBUFTYPE`, and `MAXBUFSTYPE` Parameters
- Setting the `SANITYSCAN`, `BLOCKTIME`, `BBLQUERY`, and `DBBLWAIT` Parameters

You can set these application parameters to enhance the efficiency of your system.

Setting the MAXACCESSERS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES Parameters

The MAXACCESSERS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES parameters increase semaphore and shared memory costs, so you should choose the minimum value that satisfies the needs of the system. You should also allow for the variation in the number of clients accessing the system at the same time. Defaults may be appropriate for a generous allocation of IPC resources. However, it is prudent to set these parameters to the lowest appropriate values for the application.

For multithreaded WebLogic Enterprise Java servers, you must account for the number of worker threads that each server is configured to run. The MAXACCESSERS parameter sets the maximum number of concurrent accessors of a WebLogic Enterprise system. Accessors include native and remote clients, servers, and administration processes.

A single-threaded server counts as one accessor. For a multithreaded Java server, the number of accessors can be up to twice the maximum number of worker threads that the server is configured to run, plus one for the server itself. However, to calculate a MAXACCESSERS value for a WebLogic Enterprise system running multithreaded servers, *do not* simply double the existing MAXACCESSERS value of the whole system. Instead, you add up the accessors for each multithreaded server.

For example, assume that your system has three multithreaded Java servers:

- Java server A is configured to run three worker threads.
- Java server B is configured to run four worker threads.
- Java server C is configured to run five worker threads.

The accessor requirement of these servers is calculated by using the following formula:

$$[(3 * 2) + 1] + [(4 * 2) + 1] + [(5 * 2) + 1] = 27 \text{ accessors}$$

Setting the MAXGTT, MAXBUFTYPE, and MAXBUFSTYPE Parameters

You should increase the value of the MAXGTT parameter if the product of multiplying the number of clients in the system times the percentage of time they are committing a transaction is close to 100. This may require a great number of clients, depending on the speed of commit. If you increase MAXGTT, you should also increase TLOGSIZE accordingly for every machine. You should set MAXGTT to 0 for applications that do not use distributed transactions.

You can limit the number of buffer types and subtypes allowed in the application with the MAXBUFTYPE and MAXBUFSTYPE parameters, respectively. The current default for MAXBUFTYPE is 16. Unless you are creating many user-defined buffer types, you can omit MAXBUFTYPE. However, if you intend to use many different VIEW subtypes, you may want to set MAXBUFSTYPE to exceed its current default of 32.

Setting the SANITYSCAN, BLOCKTIME, BBLQUERY, and DBBLWAIT Parameters

If a system is running on slower processors (for example, due to heavy usage), you can increase the timing parameters: SANITYSCAN, BLOCKTIME, and individual transaction timeouts. If networking is slow, you can increase the value of the BLOCKTIME, BBLQUERY, and DBBLWAIT parameters.

Setting Application Parameters

Table 6-3 describes the system parameters available for tuning an application.

Table 6-3 System Parameters for Application Tuning

Parameters	Action
MAXACCESSERS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES	Set the smallest satisfactory value because of IPC cost. Allow for extra clients.
MAXGTT, MAXBUFTYPE, and MAXBUFSTYPE	Increase MAXGTT for many clients; set MAXGTT to 0 for nontransactional applications. Use MAXBUFTYPE only if you create eight or more user-defined buffer types. If you use many different VIEW subtypes, increase the value of MAXBUFSTYPE.
BLOCKTIME, TRANTIME, and SANITYSCAN	Increase the value for a slow system.
BLOCKTIME, TRANTIME, BBLQUERY, and DBBLWAIT	Increase values for slow networking.

Determining IPC Requirements

The values of different system parameters determine IPC requirements. You can use the `tmboot -c` command to test a configuration’s IPC needs. The values of the following parameters affect the IPC needs of an application:

- `MAXACCESSERS`
- `REPLYQ`
- `RQADDR` (that allows MSSQ sets to be formed)
- `MAXSERVERS`
- `MAXSERVICES`
- `MAXGTT`

Table 6-4 describes the system parameters that affect the IPC needs of an application.

Table 6-4 Tuning IPC Parameters

Parameter(s)	Action
<code>MAXACCESSERS</code>	Equals the number of semaphores. Number of message queues is almost equal to <code>MAXACCESSERS</code> + the number of servers with reply queues (the number of servers in MSSQ set + the number of MSSQ sets).
<code>MAXSERVERS</code> , <code>MAXSERVICES</code> , and <code>MAXGTT</code>	While <code>MAXSERVERS</code> , <code>MAXSERVICES</code> , <code>MAXGTT</code> , and the overall size of the <code>ROUTING</code> , <code>GROUP</code> , and <code>NETWORK</code> sections affect the size of shared memory, an attempt to devise formulas that correlate these parameters can become complex. Instead, simply run <code>tmboot -c</code> or <code>tmloadcf -c</code> to calculate the minimum IPC resource requirements for your application.

Table 6-4 Tuning IPC Parameters (Continued)

Parameter(s)	Action
Queue-related kernel parameters	<p>Need to be tuned to manage the flow of buffer traffic between clients and servers. The maximum total size of a queue in bytes must be large enough to handle the largest message in the application, and to typically be 75 to 85 percent full. A smaller percentage is wasteful. A larger percentage causes message sends to block too frequently.</p> <p>Set the maximum size for a message to handle the largest buffer that the application sends.</p> <p>Maximum queue length (the largest number of messages that are allowed to sit on a queue at once) must be adequate for the application's operations.</p> <p>Simulate or run the application to measure the average fullness of a queue or its average length. This may be a trial and error process in which tunables are estimated before the application is run and are adjusted after running under performance analysis.</p> <p>For a large system, analyze the effect of parameter settings on the size of the operating system kernel. If unacceptable, reduce the number of application processes or distribute the application to more machines to reduce MAXACCESSERS.</p>

Measuring System Traffic

This topic includes the following sections:

- About System Traffic and Bottlenecks
- Example of Detecting a System Bottleneck
- Detecting Bottlenecks on UNIX
- Detecting Bottlenecks on Windows NT

For more information about monitoring WebLogic Enterprise applications and measuring traffic, see “Monitoring a Running System” in the *Administration Guide*.

About System Traffic and Bottlenecks

Bottlenecks can occur in your system when traffic volume nears resource capacity. You can measure service traffic using a global counter in your implementation code.

For example, in Tuxedo applications, when `tpsvrinit()` is invoked at boot time, you can initialize a global counter and record a starting time. Subsequently, each time a particular service is called, the counter is incremented. When the server is shut down by invoking the `tpsvrdone()` function, the final count and the ending time are recorded. This mechanism allows you to determine how busy a particular service is over a specified period of time.

Note: For CORBA C++ applications, use the `Server::initialize()` and `Server::release()` operations. For CORBA Java applications, use the `Server.initialize` and `Server.release` methods.

In BEA Tuxedo, bottlenecks can originate from data flow patterns. The quickest way to detect bottlenecks is to begin with the client and measure the amount of time required by relevant services.

Example of Detecting a System Bottleneck

Suppose Client 1 requires 4 seconds to print to the screen. Calls to `time(2)` determine that the `tpcall` to service A is the culprit with a 3.7 second delay. Service A is monitored at the top and bottom and takes 0.5 seconds. This implies that a queue may be clogged, which was determined by using the `pq` command.

On the other hand, suppose service A takes 3.2 seconds. The individual parts of Service A can be bracketed and measured. Perhaps Service A issues a `tpcall` to Service B, which requires 2.8 seconds. It should then be possible to isolate queue time or message send blocking time. Once the relevant amount of time has been identified, the application can be retuned to handle the traffic.

Using `time(2)`, you can measure the duration of the following:

- The entire client program.
- A client service request only.
- The entire service function.

- The service function making a service request (if any).

Detecting Bottlenecks on UNIX

On UNIX systems, the `sar(1)` command provides valuable performance information that can be used to find system bottlenecks. You can use the `sar(1)` command to:

- Sample cumulative activity counters in the operating system at predetermined intervals.
- Extract data from a system file.

Table 6-5 describes the `sar(1)` command options.

Table 6-5 `sar(1)` Command Options

Option	Description
-u	Gathers CPU utilization numbers, including the portion of the time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle.
-b	Reports buffer activity, including transfers per second of data between system buffers and disk, or other block devices.
-c	Reports system call activity. This includes system calls of all types, as well as specific system calls such as <code>fork(2)</code> and <code>exec(2)</code> .
-w	Monitors system swapping activity. This includes the number of transfers for swap-ins and swap-outs.
-q	Reports average queue lengths while occupied and the percent of time occupied.
-m	Reports message and system semaphore activities, including the number of primitives per second.
-p	Reports paging activity, including the address translation page faults, page faults and protection errors, and the valid pages reclaimed for free lists.

Table 6-5 sar(1) Command Options (Continued)

Option	Description
-r	Reports unused memory pages and disk blocks, including the average number of pages available to user processes and the disk blocks available for process swapping.

Note: Some UNIX platforms do not provide the `sar(1)` command, but offer equivalent commands instead. BSD, for example, offers the `iostat(1)` command. Sun offers `perfmeter(1)`.

Detecting Bottlenecks on Windows NT

On Windows NT, use the Performance Monitor to collect system information and detect bottlenecks. Click the Start button and select Programs, then Administration Tools, and then click NT Performance Monitor.

Index

A

- Application Assemblers, scaling tasks 4-5
- application parameters
 - setting 6-14
 - using 6-11
- application scalability requirements 1-2
- AUTOTRAN parameter 5-9

B

- BBLQUERY parameter 6-13, 6-14
- BLOCKTIME parameter 6-13, 6-14
- bottlenecks, detecting 6-17
- bundling services
 - about bundling services 6-10
 - when to bundle services 6-10

C

- CacheFullException 4-7
- caching
 - configuring the EJB cache 4-7
 - EJBCACHE_FLUSH parameter 4-10
 - MAXEJBCACHE parameter 4-10
 - tuning the EJB cache 4-9
- CLOSEINFO parameter 5-8
- connection pooling 4-3
- create_object_reference() operation 2-15
- customer support contact information xi

D

- data-dependent routing
 - about data-dependent routing 5-4
 - characteristics 5-5
 - sample application 5-5
 - using (Tuxedo only) 5-4
- DBBLWAIT parameter 6-13, 6-14
- Deployers, scaling tasks 4-5
- distributing applications
 - about distributing applications 5-2
 - benefits 5-2
 - characteristics of a distributed application 5-3
 - domain gateway file and routing 5-11
 - factory-based routing in multiple domains 5-11
 - sample application 5-5
 - UBBCONFIG file 5-10
- DMCONFIG file
 - about the DMCONFIG file 5-12
 - DM_ROUTING section 5-12
 - example 5-15
- documentation, where to find it x
- domain gateway configuration file (DMCONFIG) 5-11

E

- EJB cache 4-7
- EJB Providers

- bean persistence 4-3
- pooled connections, using 4-3
- process-entity design pattern 4-5
- scalling tasks 4-2
- stateful session beans, minimizing 4-3
- stateless session beans, using 4-2
- transactions, completing efficiently 4-4
- EJBCACHE_FLUSH parameter 4-10
- entity beans
 - persistence 4-3
 - process-entity design pattern 4-5

F

- factory_finder.ini 5-11
- factory-based routing
 - about factory-based routing 1-16
 - characteristics of 1-17
 - configuring 1-19
 - in JDBC Bankapp sample application 3-11
 - in Production sample application 2-12
 - configuring for multiple domains 5-11
 - how it works 1-18
 - implementing in a factory 2-15, 3-12
 - in JDBC Bankapp sample application 3-10
 - in Production sample application 2-11
- file-based persistence 4-9

G

- GROUP parameter 6-6
- GRPNO parameter 5-7

I

- IIOP Handler (ISH)
 - about the ISH 1-20
 - increasing the number of ISH processes

- 1-20
- IIOP Listener (ISL) 1-20, 4-7
- interfaces, assigning priorities to 6-9
- iostat(1) command 6-19
- IPC requirements
 - determining 6-15–6-16
 - tuning parameters 6-15
 - tuning queue-related kernel parameters 6-16
- is-modified-method-name element 4-7

J

- JDBC Bankapp sample application
 - additional design considerations 3-14
 - design goals 3-2
 - factory-based routing 3-10
 - how it has been scaled 3-2
 - object state management 3-3
 - replicating server groups 3-6
 - replicating server processes 3-4
 - scaling the application further 3-16
 - UBBCONFIG file 3-7
- JDBC connection pooling 4-3

K

- kernel parameters, tuning 6-16

L

- LMID parameter 5-7
- load balancing
 - about load balancing 6-4
 - enabling 6-4
 - measuring service performance time 6-5

M

- MAX parameter 6-6
- MAXACCESSERS parameter 6-12, 6-14, 6-15

- MAXBUFSTYPE parameter 6-13, 6-14
- MAXBUFTYPE parameter 6-13, 6-14
- MAXEJBACACHE parameter 4-10
- MAXGTT parameter 6-13, 6-14, 6-15
- MAXINTERFACES parameter 6-12, 6-14
- MAXSERVERS parameter 6-12, 6-14, 6-15
- MAXSERVICES parameter 6-12, 6-14, 6-15
- method-bound objects 1-5
- MIN parameter 6-6
- MSSQ sets
 - example 6-3
 - using 6-2
- multiple server single queue (MSSQ) 6-2
- multiplexing incoming client connections 1-19
- multithreading
 - about multithreaded Java servers 1-13
 - coding recommendations 1-15
 - configuring
 - number of concurrent accessors 6-8
 - number of threads 6-7
 - OPENINFO parameter 6-7
 - when to use 1-14

O

- object state management
 - in JDBC Bankapp sample application 3-3
 - in Production sample application 2-4
- object state models
 - CORBA applications 1-4
 - EJB applications 1-6
 - RMI applications 1-6
- object state models
 - stateful objects 1-6
 - stateless objects 1-6
- objects
 - method-bound 1-5
 - process-bound 1-5
 - stateful objects 1-7
 - stateless objects 1-7
 - transaction-bound 1-5
- OMG IDL, Production sample application 2-4
- OPENINFO parameter 5-8

P

- perfmer(1) command 6-19
- performance time, servopts(5) -r option 6-5
- performance, measuring 6-5
- persistence
 - file-based persistence 4-9
 - implementing methods 4-3
 - is-modified-method-name element 4-7
 - persistence-store-directory-root element 4-6
 - removing orphan files 4-8
- persistence-store-directory-root element 4-6
- pooled connections 4-3
- printing product documentation x
- PRIO parameter 6-9
- priorities
 - assigning to interfaces or services 6-9
 - PRIO parameter 6-9
- process-bound objects 1-5
- process-entity design pattern 4-5
- Production sample application
 - additional design considerations 2-17
 - changing the OMG IDL 2-4
 - design goals 2-2
 - factory-based routing 2-11
 - how it has been scaled 2-2
 - replicating server groups 2-8
 - replicating server processes 2-6
 - scaling the application further 2-22
 - stateless object model 2-4
 - UBBCONFIG file 2-9

R

related information xi

replicating

- about replicating server processes and server groups 1-10

- configuration options 1-11

- server groups

 - about replicating server groups 1-12

 - in JDBC Bankapp sample

 - application 3-6

 - in Production sample application 2-8

- server processes

 - about replicating server processes 1-11

 - benefits of 1-11

 - guidelines for 1-12

 - JDBC Bankapp sample application 3-4

 - Production sample application 2-6

resources, maximizing application 6-2–6-14

ROUTING parameter 5-9

S

SANITYSCAN parameter 6-13, 6-14

sar(1) command 6-18

scalability

- features 1-2

- requirements 1-2

- support, in WLE applications 1-3

scaling tasks

- Application Assemblers 4-5

- Deployers 4-5

- EJB Providers 4-2

- System Administrators 4-8

server groups

- about replicating 1-10

- replicating 1-12

server processes

- about replicating 1-10

- replicating 1-11

SERVICES section

- EJBCACHE_FLUSH parameter 4-10

- MAXEJBCACHE parameter 4-10

servopts(5) 6-5

SRVID parameter 6-6

stateful objects

- about stateful objects 1-7

- when to use 1-8

stateful session beans

- minimizing state information 4-3

- persistence 4-3

stateless objects

- about stateless objects 1-7

- when to use 1-7

stateless session beans

- using 4-2

support

- technical xi

System Administrators, scaling tasks for 4-8

T

time(2) option 6-5

tmboot(1) -c command 6-15

TMSCOUNT parameter 5-8

TMSNAME parameter 5-7

traffic, measuring system 6-16–6-19

transaction-bound objects 1-5

transactions, in EJB applications 4-4

TRANTIME parameter 5-9, 6-14

tsprio call 6-9

tuning applications 6-1–6-19

- determining IPC requirements 6-15

- maximizing application resources 6-2

 - bundling services into servers 6-10

 - enabling load balancing 6-4

- measuring system traffic 6-16

 - detecting a system bottleneck 6-17

- using application parameters 6-11, 6-12, 6-13

U

UBBCONFIG file

- distributed application example 5-10

GROUPS section

- CLOSEINFO parameter 5-8

- GRPNO parameter 5-7

- LMID parameter 5-7

- OPENINFO parameter 5-8, 6-7

- TMSCOUNT parameter 5-8

- TMSNAME parameter 5-7

- in JDBC Bankapp sample application 3-7

- in Production sample application 2-9

ROUTING section 5-10

SERVERS section

- GROUP parameter 6-6

- MAX parameter 6-6

- MIN parameter 6-6

- SRVID parameter 6-6

SERVICES section

- AUTOTRAN parameter 5-9

- ROUTING parameter 5-9

- sample 5-9

- TRANTIME parameter 5-9