BEA WebLogic Server™

Using WebLogic Server Clusters

BEA WebLogic Server Version 6.1
Document Date: October 20, 2003

Copyright

Copyright © 2002 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. 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 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, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Collaborate, BEA WebLogic Commerce Server, BEA WebLogic E-Business Platform, BEA WebLogic Enterprise, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Process Integrator, BEA WebLogic Server, E-Business Control Center, How Business Becomes E-Business, Liquid Data, Operating System for the Internet, and Portal FrameWork are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Using WebLogic Server Clusters

Part Number	Document Date	Software Version
N/A	June 24, 2002	BEA WebLogic Server Version 6.1

Contents

About This Document

1. Introduction to WebLogic Server Clustering

2. Cluster Features and Infrastructure

3. Understanding HTTP Session State Replication

4. Understanding Object Clustering

5. Planning WebLogic Server Clusters

7. Troubleshooting Common Problems

A. The WebLogic Cluster API

B. Configuring BIG-IP™ Hardware with Clusters

About This Document

This document describes BEA WebLogic Server™ Clusters, and provides an introduction to developing clusters with WebLogic Server 6.1.

The document is organized as follows:

- Chapter 1, "Introduction to WebLogic Server Clustering," introduces WebLogic Server Cluster concepts, and summarizes the changes to clustering in WebLogic Server 6.1.

- Chapter 2, "Cluster Features and Infrastructure," describes the basic features that a cluster provides for HTTP sessions and clustered objects.

- Chapter 3, "Understanding HTTP Session State Replication," explains how a WebLogic Server cluster replicates HTTP session states in memory to provide automatic load balancing and failover.

- Chapter 4, "Understanding Object Clustering," describes how WebLogic Server provides load balancing and failover for clustered EJBs and RMI objects.

- Chapter 5, "Planning WebLogic Server Clusters," describes common issues to consider before deploying one or more WebLogic Server clusters. This section also presents recommended cluster architectures for common Web applications.

- Chapter 6, "Administering WebLogic Clusters," introduces WebLogic Server Cluster administration, including information on how to set up and run a WebLogic Server Cluster.

- Chapter 7, "Troubleshooting Common Problems," provides a checklist to assist you in resolving cluster problems.

- "The WebLogic Cluster API" introduces the cluster API for RMI objects and offers information on development using the API.

- "Configuring BIG-IP™ Hardware with Clusters" contains instructions for configuring the BIG-IP load balancer.

Audience

This document is written for application developers and administrators who are interested in deploying Web-based applications onto one or more clusters. It is assumed that readers have a familiarity with HTTP, HTML coding, and Java programming (servlets, JSP, or EJB development).

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

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 Server 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 Server 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, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

- The WebLogic Server EJB Container section of the *Programming WebLogic Enterprise JavaBeans* manual at http://e-docs.bea.com/wls/docs61/ejb/EJB_environment.html

- Programming WebLogic HTTP Servlets at http://e-docs.bea.com/wls/docs61/servlet/index.html

- Deploying and Configuring Web Applications section of the *WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/config_web_app.html

Contact Us!

Your feedback on the BEA WebLogic Server 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 Server documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation.

If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSUPPORT at http://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.
italics	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.
	Examples:
	`#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. *Example*: `void commit ( )`
`monospace italic text`	Identifies variables in code. *Example*: `String expr`

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
{ }	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. *Example*: ```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: - 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. *Example*: ```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 Introduction to WebLogic Server Clustering

The following sections introduce WebLogic Server clusters:

- What Is a WebLogic Server Cluster?

- What Services Are Clustered?

- New Cluster Features in WebLogic Server Version 6.1

What Is a WebLogic Server Cluster?

A WebLogic Server cluster is a group of servers that work together to provide a more scalable, more reliable application platform than a single server. A cluster appears to its clients as a single server but is in fact a group of servers acting as one. A cluster provides two key features above a single server:

- **Scalability**: The capacity of a cluster is not limited to a single server or a single machine. New servers can be added to the cluster dynamically to increase capacity. If more hardware is needed, a new server on a new machine can be added. If a single server cannot fully utilize an existing machine, additional servers can be added to that machine.

■ **High-Availability**: A cluster uses the redundancy of multiple servers to insulate clients from failures. The same service can be provided on multiple servers in the cluster. If one server fails, another can take over. The ability to failover from a failed server to a functioning server increases the availability of the application to clients.

WebLogic Server clusters are designed to bring scalability and high availability to J2EE applications. They provide these features in a way that is transparent to the application writer and to clients. It is important, however, for application programmers and administrators to understand the issues inherent in clustering in order to maximize the scalability and availability of their applications.

What Services Are Clustered?

A clustered service is an API or interface that is available on multiple servers in the cluster. HTTP session state clustering and object clustering are the two primary cluster services that WebLogic Server provides. WebLogic Server also provides cluster support for JMS destinations and JDBC connections, as described in the sections that follow.

HTTP Session States

WebLogic Server provides clustering support for servlets and JSPs by replicating the HTTP session state of clients that access clustered servlets and JSPs. To benefit from HTTP session state clustering, you must ensure that the session state is persistent, either by configure in-memory replication, filesystem persistence, or JDBC persistence. "Understanding HTTP Session State Replication" on page 3-1 describes in-memory replication for clustered servlet and JSPs. See "Making Sessions Persistent" in *Programming WebLogic HTTP Servlets* for more information about file persistence and JDBC persistence.

EJBs and RMI objects

Load balancing and failover for EJBs and RMI objects is handled using special, *replica-aware stubs*, which can locate instances of the object throughout the cluster. You create replica-aware stubs for EJBs and RMI objects by specifying the appropriate deployment descriptors or providing command-line options to rmic. When you deploy a clustered object, you deploy it to all the server instances in the cluster (homogeneous deployment). "Understanding Object Clustering" on page 4-1 describes object clustering in more detail.

JMS

The WebLogic JMS architecture implements clustering of multiple JMS servers by supporting cluster-wide, transparent access to destinations from any server in the cluster. Although WebLogic Server supports distributing JMS destinations and connection factories throughout a cluster, JMS Queues and Topics are still managed by individual WebLogic Server instances in the cluster.

The advantages of clustering WebLogic JMS include:

■ Establish cluster-wide, transparent access to destinations from any server in the cluster by configuring multiple connection factories and using targets to assign them to WebLogic Servers. Each connection factory can be deployed on multiple WebLogic Servers.

■ Load balance destinations across multiple servers instances in the cluster by configuring multiple JMS servers and using targets to assign them to the defined WebLogic Servers. Each JMS server is deployed on exactly one WebLogic Server instance and handles requests for a set of destinations.

For more information about WebLogic JMS and cluster support for JMS, see "Introduction to WebLogic JMS" in *Programming WebLogic JMS*.

JDBC Connections

WebLogic Server allows you to cluster JDBC objects, including data sources, connection pools and multipools, to improve the availability of cluster-hosted applications. Each JDBC object you configure for your cluster must exist on *each* managed server in the cluster—when you configure the JDBC objects, target them to the cluster.

- Data Sources—In a cluster, external clients must obtain connections through a JDBC data source on the JNDI tree. The data source uses the WebLogic Server RMI driver to acquire a connection. The cluster-aware nature of WebLogic data sources in external client applications allows a client to request another connection if the server instance hosting the previous connection fails. Although not strictly required, BEA recommends that server-side clients also obtain connections via a data source on the JNDI tree.

- Connection Pools—Connection pools are a collection of ready-to-use database connections. When a connection pool starts up, it creates a specified number of identical physical database connections. By establishing connections at start-up, the connection pool eliminates the overhead of creating a database connection for each application. BEA recommends that both client and server-side applications obtain connections from a connection pool through a Data Source on the JNDI tree. When finished with a connection, applications return the connection to the connection pool.

- Multipools—Multipools are multiplexers for basic connection pools. To the application they appear exactly as basic pools, but multipools allow you to establish a pool of connection pools, in which the connection attributes vary from connection pool to connection pool. All of the connections in a given connection pool are identical, but the connections in each connection pool in a multipool should vary in some significant way such that an expected failure of one pool will not invalidate another pool in the multipool. Usually these pools will be to different instances of the same database.

  Multipools are only useful if there are multiple distinct database instances that can equally handle an application connection, and the application system takes care of synchronizing the databases when application work is distributed among the databases. In rare cases it may be valuable to have the pools to the same database instance, but as different users. This would be useful if the DBA disabled one user, leaving the other user viable.

By default, a clustered multipool provides high availability (DBMS failover). A multipool can be optionally configured to also provide load balancing.

For more information about JDBC, see "JDBC Components—Connection Pools, Data Sources, and MultiPools" in the *WebLogic Server Administration Guide.*

Getting Connections with Clustered JDBC

To ensure that any JDBC request can be handled equivalently by any cluster member, each managed server in the cluster must have similarly named/defined pools and, if applicable, multipools. Data sources, if intended for use in external clients, should be targeted to the cluster so they are cluster-aware and their connections can be to any cluster members.

- External Clients Connections—External clients that require a database connection perform a JNDI lookup and obtain a replica-aware stub for the data source. The stub for the data source contains a list of the server instances that host the data source—which should be all of the Managed Servers in the cluster. Replica-aware stubs contain load balancing logic for distributing the load among host server instances.

- Server-Side Client Connections—For server-side use, application code can use a server-side pool driver directly instead of a JNDI lookup and a data source, but if a data source is used, it will be a local object. A server-side data source will not go to another cluster member for its JDBC connections. The data source obtains a connection from the pool it references. The connection is pinned to the local server instance for the duration of the database transaction, and as long as the application code retains it (until the connection is closed).

Failover and Load Balancing for JDBC Connections

Clustering your JDBC objects does not enable failover of connections but it can ease the process of reconnection when a connection fails. In replicated database environments, multipools may be clustered to support database failover, and optionally, load balancing of connections. See the following topics for more information:

- To understand the behavior of clustered JDBC objects when failures occur, see "Failover and JDBC Connections" on page 2-21.

- To learn more about how clustered multipools enable load balancing of connections, see "Load Balancing and JDBC Connections" on page 2-18.

Non-Clustered Services and APIs

Several APIs and internal services cannot be clustered in WebLogic Server. These include:

- File services

- Time services

- WebLogic Events (deprecated in WebLogic Server 6.0)

- Workspaces (deprecated in WebLogic Server 6.0)

- ZAC

You can still use these services on individual WebLogic Server instances in a cluster. However, the services do not make use of load balancing or failover features.

New Cluster Features in WebLogic Server Version 6.1

WebLogic Server version 6.1 introduces the following new features and improvements when configured as a cluster of server instances, compared to Weblogic Server version 5.x.

Integrated Support for Load Balancing Hardware

WebLogic Server now supports load balancing and failover for clustered servlets and JSPs when clients connect directly to a cluster via supported load balancing hardware. Clustered systems are no longer required to proxy HTTP requests to a cluster using the `HttpClusterServlet` or WebLogic Server proxy plug-ins. Understanding HTTP Session State Replication describes this in more detail.

Stateful Session EJB Clustering

WebLogic Server now supports clustering the `EJBObject` for stateful session EJB instances (as well as the `EJBHome` object). WebLogic Server replicates the state of stateful session EJBs in memory, in a manner similar to HTTP session state replication. See Understanding Object Clustering for more information.

Clustered JMS

WebLogic Server now supports distributing JMS destinations and connection factories throughout a cluster. JMS topics and queues, however, are still managed by individual WebLogic Server instances in the cluster.

HTTP Session State Replication Changes

WebLogic Server version 6.1 introduces a more robust mechanism for replicating servlet session states across clustered server instances. As with previous server versions, WebLogic Server maintains two copies of the servlet session state (a primary and secondary) on different server instances in the cluster. Version 6.1 improves the replication system by:

- Providing administrative control over where WebLogic Server creates session state replicas.

- Enabling clients to failover to a secondary session state regardless of where in the WebLogic Server cluster those clients connect.

These changes provide more opportunities for addressing failover scenarios, and also make it possible for WebLogic Server to operate directly in conjunction with load balancing hardware. See Understanding HTTP Session State Replication for more information.

Administration Changes in WebLogic Server Version 6.1

The following changes affect the way you administer clusters in WebLogic Server 6.1, compared to WebLogic Server version 5.x.

Multicast Message Changes

Multiple WebLogic Server version 6.1 clusters can now "share" a single multicast address without causing broadcast message conflicts. You do not need to assign a dedicated multicast address for each cluster.

You should still ensure that no other applications utilize the WebLogic Server multicast address. When other applications broadcast on the cluster multicast address, all servers in the cluster must deserialize those messages to determine that they are not cluster-related messages. This introduces unnecessary overhead, and may cause servers to miss actual cluster broadcast messages.

Homogeneous Deployment

WebLogic Server version 6.1 supports only homogeneous deployment of clustered objects. If an object is compiled as clusterable (if it has a replica-aware stub), you must deploy the object to all members of the cluster.

Non-clustered objects (EJBs and RMI classes) may deployed to individual servers in the cluster.

Administration Server Configuration

You perform all configuration for a WebLogic Server 6.1 cluster using the Administration Console. The Administration Console stores all configuration information for the cluster in a single XML configuration file. The Administration Console also manages deployment of objects and Web applications to members of the cluster.

For general information about configuring WebLogic Server 6.1, see the Administration Guide. See Administering WebLogic Clusters for specific instructions about using the Administration Console to configure a cluster.

2 Cluster Features and Infrastructure

The following sections describe the infrastructure that a WebLogic Server cluster uses to support clustered objects and HTTP session states:

■ Overview

■ Server Communication in a Cluster

■ Cluster-Wide JNDI Naming Service

■ Load Balancing of Clustered Services

■ Failover Support for Clustered Services

Overview

This section also describes the common features—load balancing and failover—that are available to APIs and services running in a WebLogic Server cluster. Understanding these topics is important for planning and configuring a WebLogic Server cluster that meets the needs of your Web application.

Server Communication in a Cluster

WebLogic Server instances in a cluster communicate with one another using two basic network technologies:

- IP multicast, which broadcasts all one-to-many communications among clustered WebLogic Server instances.

- IP sockets, which act as the conduits for peer-to-peer communication between clustered server instances.

The way in which WebLogic Server uses IP multicast and socket communication has a direct implication on the way you plan and configure your cluster.

One-to-Many Communication Using IP Multicast

IP multicast is a simple broadcast technology that enables multiple applications to "subscribe" to a given IP address and port number and listen for messages. A multicast address is an IP address in the range from 224.0.0.0 to 239.255.255.255.

IP multicast provides a simple method to broadcast messages to applications, but it does not guarantee that messages are actually received. If an application's local multicast buffer is full, new multicast messages cannot be written to the buffer and the application is not notified as to when messages are "dropped." Because of this limitation, WebLogic Servers account for the possibility that they may occasionally miss messages that were broadcast over IP multicast.

WebLogic Server uses IP multicast for all one-to-many communication among server instances in a cluster. This includes:

- Cluster-wide JNDI updates—all servers use multicast to announce the availability of clustered objects that are deployed or removed locally. Servers monitor these announcements so that they can update their local JNDI tree to reflect current deployments of clustered objects. See "Cluster-Wide JNDI Naming Service" on page 2-10 for more details.

- Cluster heartbeats— WebLogic Server uses multicast to broadcast regular "heartbeat" messages that advertise the availability of individual server instances in a cluster. All servers in the cluster listen to heartbeat messages as a way to

determine when a server has failed. (Clustered servers also monitor IP sockets as a more immediate method of determining when a server has failed.) See Failover Support for Clustered Services for more information.

Implications for Cluster Planning and Configuration

Because multicast controls critical functions related to detecting failures and maintaining the cluster-wide JNDI tree, it is important that neither the cluster configuration nor the basic network topology interfere with multicast communication. Always consider the following rules when configuring or planning a WebLogic Server cluster.

Multicast Requirements for WAN Clustering

For most deployments, limiting clustered servers to a single subnet ensures that multicast messages are reliably transmitted. In special cases, however, you may want to distribute a WebLogic Server cluster across subnets in a Wide Area Network (WAN). This may be desirable to increase redundancy in a clustered deployment, or to distribute clustered instances over a larger geographical area.

If you choose to distribute a cluster over a WAN (or across multiple subnets), you must plan and configure your network topology to ensure that multicast messages are reliably transmitted to all servers in the cluster. Specifically, your network must meet the following requirements:

■ The network must fully support IP multicast packet propagation. In other words, all routers and other tunneling technologies must be configured to propagate multicast messages to clustered instances.

■ The network latency must be sufficiently small as to ensure that most multicast messages reach their final destination in 200 to 300 milliseconds.

■ The multicast Time-To-Live (TTL) value must be high enough to ensure that routers do not discard multicast packets before they reach their final destination. For instructions on setting multicast TTL, see "Configure Multicast Time-To-Live (TTL)" on page 6-23.

Note: Distributing a WebLogic Server cluster over a WAN may require network facilities in addition to the multicast requirements described above. For example, you may want to configure load balancing hardware to ensure that client requests are directed to servers in the most efficient manner (to avoid unnecessary network hops).

Firewalls Can Break Multicast Communication

Although it may be possible to tunnel multicast traffic through a firewall, this practice is not recommended for WebLogic Server clusters. Each WebLogic Server cluster should be treated as a logical unit that provides one or more distinct services to clients of a Web application. Such a logical unit should not be split between different security zones. Furthermore, any technologies that can potentially delay or interrupt IP traffic can prove disruptive to a WebLogic Server cluster by generating false failures due to missed heartbeats.

Use an Exclusive Multicast Address for WebLogic Server Clusters

Although multiple WebLogic Server clusters can share a single IP multicast address and port number, other applications should not broadcast or subscribe to the same address. "Sharing" a multicast address with other applications forces clustered servers to process unnecessary messages, introducing overhead to the system.

Sharing a multicast address may also overload the IP multicast buffer and delay transmission of WebLogic Server heartbeat messages. Such delays can potentially result in a WebLogic Server instance being marked as failed, simply because its heartbeat messages were not received in a timely manner.

For these reasons, assign a dedicated multicast address for use by WebLogic Server clusters, and ensure that the address can support the broadcast traffic of all clusters that use the address.

If Multicast Storms Occur

If server instances in a cluster do not process incoming messages on a timely basis, increased network traffic, including NAK messages and heartbeat re-transmissions, can result. The repeated transmission of multicast packets on a network is referred to as a *multicast storm*, and can stress the network and attached stations, potentially causing end-stations to hang or fail. Increasing the size of the multicast buffers can improve the rate at which announcements are transmitted and received, and prevent multicast storms.

If multicast storms occur because server instances in a cluster are not processing incoming messages on a timely basis, you can increase the size of multicast buffers.

TCP/IP kernel parameters can be configured with the UNIX ndd utility. The `udp_max_buf` parameter controls the size of send and receive buffers (in bytes) for a UDP socket. The appropriate value for `udp_max_buf` varies from deployment to deployment. If you are experiencing multicast storms, increase the value of `udp_max_buf` by 32K, and evaluate the effect of this change.

Do not change `udp_max_buf` unless necessary. Before changing `udp_max_buf`, read the Sun warning in the "UDP Parameters with Additional Cautions" section in the "TCP/IP Tunable Parameters" chapter in *Solaris Tunable Parameters Reference Manua*l at http://docs.sun.com/?p=/doc/806-6779/6jfmsfr7o&.

Peer-to-Peer Communication Using IP Sockets

While one-to-many communication among clustered servers takes place using multicast, peer-to-peer communication between WebLogic Server instances uses IP sockets. IP sockets provide a simple, high-performance mechanism for transferring messages and data between two applications. WebLogic Server instances in a cluster may use IP sockets for:

- Accessing non-clustered objects that reside on a remote server instance in the cluster.

- Replicating HTTP session states and stateful session EJB states between a primary and secondary server for high availability.

- Accessing clustered objects that reside on a remote server instance. (This generally occurs only in a multi-tier cluster architecture, as described in Recommended Multi-Tier Architecture.)

Note: The use of IP sockets in WebLogic Server actually extends beyond the cluster scenario—all RMI communication takes place using sockets, for example, when a remote Java client application accesses a remote object.

Proper socket configuration is crucial to the performance of a WebLogic Server cluster. Two factors determine the efficiency of socket communications in WebLogic Server:

- Whether the server's host system uses a native or a pure-Java socket reader implementation.

■ For systems that use Java socket readers, whether the server is configured to use enough socket reader threads.

Pure-Java Versus Native Socket Reader Implementations

Although the pure-Java implementation of socket reader threads provides a reliable and portable method of peer-to-peer communication, it does not provide the best performance for heavy-duty socket usage in a WebLogic Server cluster. With pure-Java socket readers, threads must actively poll all opened sockets to determine if they contain data to read. In other words, socket reader threads are always "busy" polling sockets, even if the sockets have no data to read.

This problem is magnified when a server has more open sockets than it has socket reader threads. In this case, each reader thread must poll more than one open socket, waiting for a timeout condition to determine that the socket is inactive. After a timeout, the thread moves to another waiting socket, as shown below.
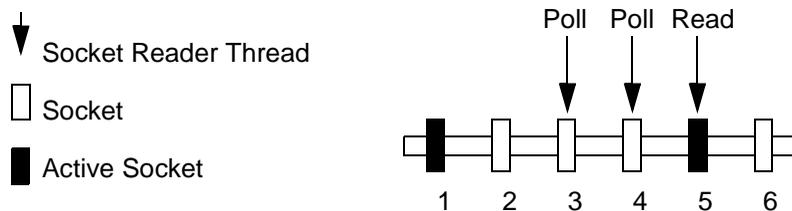
When the number of opened sockets outnumbers the available socket reader threads, active sockets may go unserviced until an available reader thread polls them.

For best socket performance, always configure the WebLogic Server host machine to use the native socket reader implementation for your operating system, rather than the pure-Java implementation. Native socket readers use far more efficient techniques to determine if there is data to read on a socket. With a native socket reader implementation, reader threads do not need to poll inactive sockets—they service only active sockets, and they are immediately notified (via an interrupt) when a given socket becomes active.

For instructions on configure the WebLogic Server host machine to use the native socket reader implementation, see "Configure Native IP Sockets Readers on Machines that Host Server Instances" on page 6-22.

Configuring Reader Threads for Java Socket Implementation

If you do use the pure-Java socket reader implementation, you can still improve the performance of socket communication by configuring the proper number of socket reader threads. For best performance, the number of socket reader threads in WebLogic Server should equal the potential maximum number of opened sockets. This avoids "sharing" a reader thread with more than one socket, and ensures that socket data is read immediately.

Determining Potential Socket Usage

Each WebLogic Server instance can *potentially* open a socket for every other server instance in the cluster. However, the actual maximum number of sockets used at a given time is determined by the configuration of your cluster. In practice, clustered systems generally do not open a socket for every other server instance, due to the way in which clustered services are deployed.

For example, if your cluster uses in-memory HTTP session state replication, and you deploy only clustered objects to all WebLogic Server instances, each server potentially opens a maximum of only two sockets, as shown below.
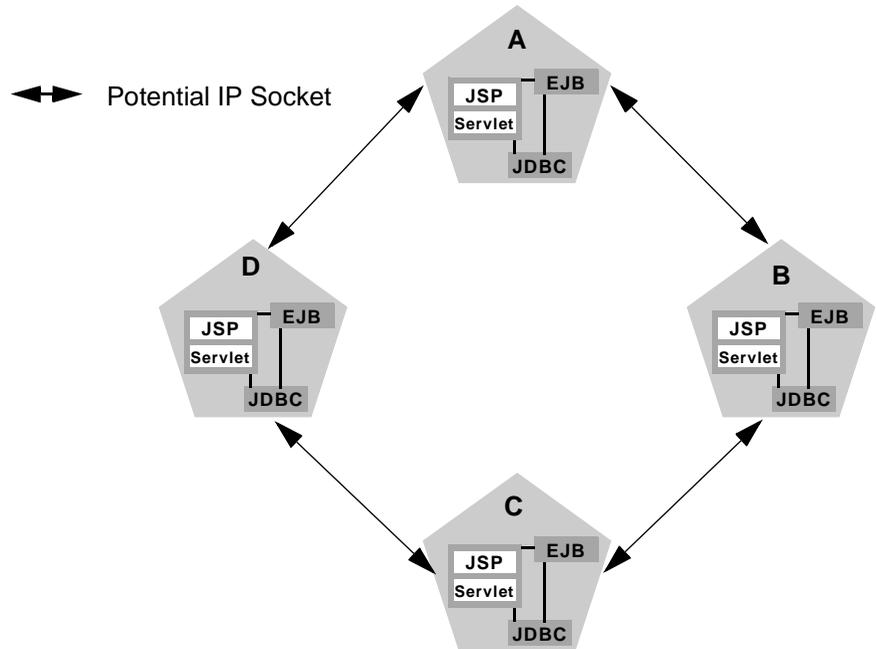
The two sockets in the above example are used to replicate HTTP session states between primary and secondary servers. Sockets are not required for accessing clustered objects, due to the collocation optimizations that WebLogic Server uses to access those objects. In this configuration, the default socket reader thread configuration is sufficient.

If you pin non-clustered RMI objects to particular servers, the potential maximum number sockets increases, because server instances may need to open additional sockets to access the pinned object. (This potential can only be released if a remote server actually looks up the pinned object.) The figure below shows the potential effect of deploying a non-clustered RMI object to Server A.
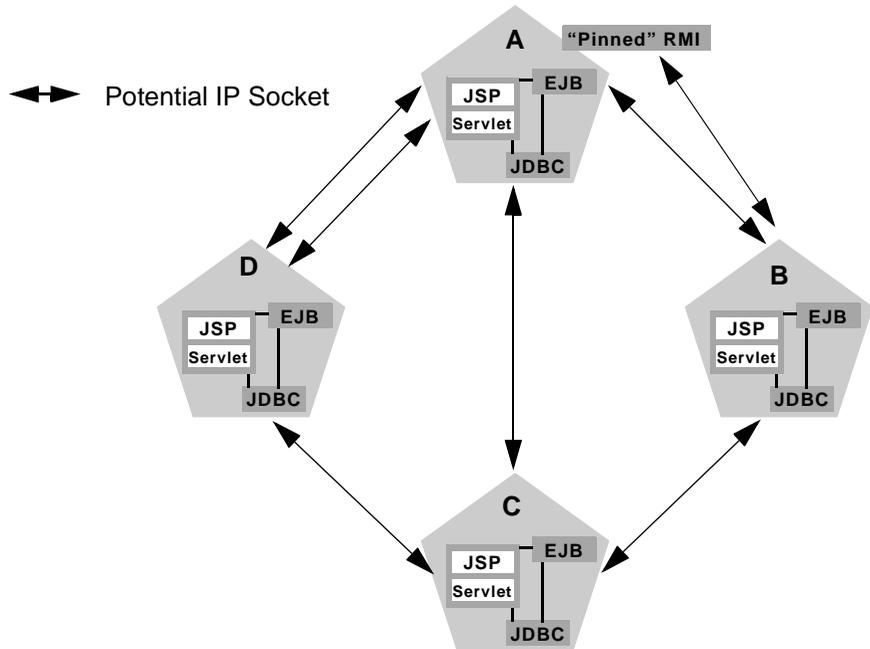
In this example, each server can potentially open a maximum of three sockets at a given time, to accommodate HTTP session state replication and to access the pinned RMI object on Server A.

Note: Additional sockets may also be required for servlet clusters in a multi-tier cluster architecture, as described in "Recommended Multi-Tier Architecture" on page 5-7.

For instructions to set the number of reader threads, see "Set the Number of Reader Threads on Machines that Host Server Instances" on page 6-22.

Client Communication via Sockets

Clients of a cluster use the Java implementation of socket reader threads. Java client applications in WebLogic Server version 6.1 can potentially open more IP sockets than clients of previous WebLogic Server versions, even when clients connect through a

firewall. Whereas in versions 4.5 and 5.1, Java clients connecting to a cluster through a firewall utilized a single socket, WebLogic Server version 6.1 imposes no such restrictions. If clients make requests of multiple server instances in a cluster (either explicitly or by accessing "pinned" objects), the client opens individual sockets to each server.

For best performance, configure enough socket reader threads in the JVM that runs the client. For instructions, see "Set the Number of Reader Threads on Client Machines" on page 6-23.

Note: Browser-based clients and Applets connecting to a WebLogic Server version 6.1 cluster use only a single IP socket.

Cluster-Wide JNDI Naming Service

Clients of an individual WebLogic Server access objects and services by using a JNDI-compliant naming service. The JNDI naming service contains a list of the public services that the server offers, organized in a "tree" structure. A WebLogic Server offers a new service by binding into the JNDI tree a name that represents the service. Clients obtain the service by connecting to the server and looking up the bound name of the service.

Server instances in a cluster utilize a cluster-wide JNDI tree. A cluster-wide JNDI tree is similar to a single server JNDI tree, insofar as the tree contains a list of available services. In addition to storing the names of local services, however, the cluster-wide JNDI tree stores the services offered by clustered objects (EJBs and RMI classes) from other servers in the cluster.

Each WebLogic Server instance in a cluster creates and maintains a local copy of the logical cluster-wide JNDI tree. By understanding how the cluster-wide naming tree is maintained, you can better diagnose naming conflicts that may occur in a clustered environment.

Warning: Do not use the cluster-wide JNDI tree as a persistence or caching mechanism for application data. Although WebLogic Server replicates a clustered server's JNDI entries to other servers in the cluster, those entries are removed from the cluster if the original server fails. Also, storing large

objects within the JNDI tree can overload multicast traffic and interfere with the normal operation of a cluster.

Creating the Cluster-Wide JNDI Tree

Each WebLogic Server in a cluster builds and maintains its own local copy of the cluster-wide JNDI tree, which lists the services offered by all members of the cluster. Creating a cluster-wide JNDI tree begins with the local JNDI tree bindings of each server instance. As a server boots (or as new services are dynamically deployed to a running server), the server first binds the implementations of those services to the local JNDI tree. The implementation is bound into the JNDI tree only if no other service of the same name exists.

Once the server successfully binds a service into the local JNDI tree, additional steps are taken for clustered objects that use replica-aware stubs. After binding a clustered object's implementation into the local JNDI tree, the server sends the object's stub to other members of the cluster. Other members of the cluster monitor the multicast address to detect when remote servers offer new services.
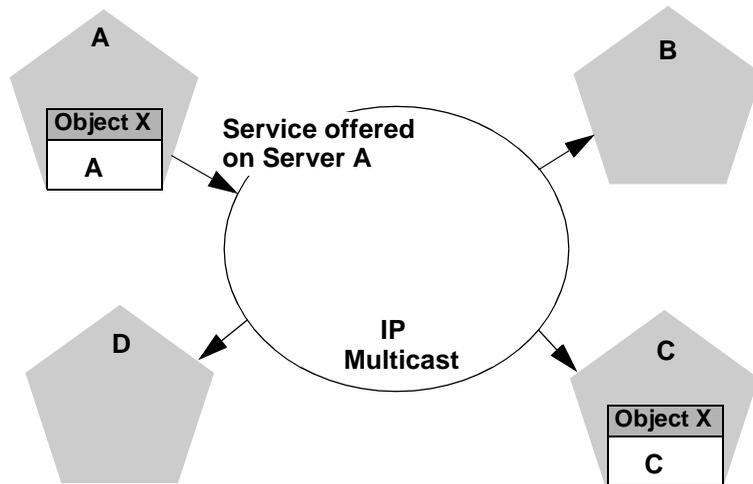
The example above shows a snapshot of the JNDI binding process. Server A has successfully bound an implementation of clustered Object X into its local JNDI tree. Because Object X is clustered, it offers this service to all other members of the cluster. Server C is still in the process of binding an implementation of Object C.

Other servers in the cluster listening to the multicast address observe that Server A offers a new service for clustered object, X. These servers update their local JNDI trees to include the new service.

Updating the local JNDI bindings occurs in one of two ways:

- If the clustered service is not yet bound in the local JNDI tree, the server binds a new replica-aware stub into the local tree that indicates the availability of Object X on Server A. Servers B and D would update their local JNDI trees in this manner, because the clustered object is not yet deployed on those servers.

- If the server already has a binding for the cluster-aware service, it updates its local JNDI tree to indicate that a replica of the service is also available on Server A. Server C would update its JNDI tree in this manner, because it will already have a binding for the clustered Object X.

In this manner, each server in the cluster creates its own copy of a cluster-wide JNDI tree. The same process would be used when Server C announces that Object X has been bound into its local JNDI tree. After all broadcast messages are received, each server in the cluster would have identical to indicate availability of the object on servers A and C, as shown below.
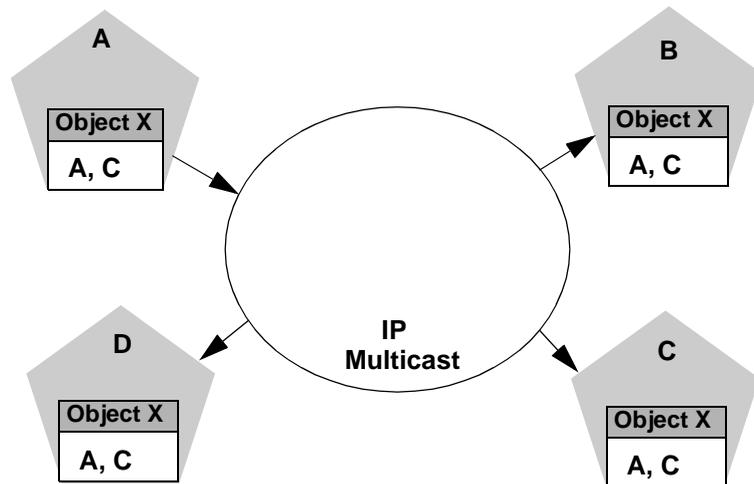
Note: In an actual cluster system, Object X would be deployed homogeneously, and an implementation would be available on all four servers.

Handling JNDI Naming Conflicts

Simple JNDI naming conflicts occur when a server attempts to bind a non-clustered service that uses the same name as a non-clustered service already bound in the JNDI tree. Cluster-level JNDI conflicts occur when a server attempts to bind a clustered object that uses the name of a non-clustered object already bound in the JNDI tree.

WebLogic Server detects simple naming conflicts (of non-clustered services) when those services are bound to the local JNDI tree. Cluster-level JNDI conflicts may occur when new services are advertised over multicast. For example, if you deploy a pinned RMI object on one server in the cluster, you cannot deploy a replica-aware version of the same object on another server instance.

If two servers in a cluster attempt to bind different clustered objects using the same name, both will succeed in binding the object locally. However, each server will refuse to bind the other server's replica-aware stub in to the JNDI tree, due to the JNDI naming conflict. A conflict of this type would remain until one of the two servers was

shut down, or until one of the servers undeployed the clustered object. This same conflict could also occur if both servers attempt to deploy a pinned object with the same name.

Homogeneous Deployment

To avoid cluster-level JNDI conflicts, you must deploy all replica-aware objects to all WebLogic Server instances in a cluster (homogeneous deployment). Having unbalanced deployments across WebLogic Server instances increases the chance of JNDI naming conflicts during startup or redeployment. It can also lead to unbalanced processing loads in the cluster.

If you must pin specific RMI objects or EJBs to individual servers, make sure you do not replicate the object's bindings across the cluster.

Updating the JNDI Tree

If a clustered object is removed (undeployed from a server), updates to the JNDI tree are handled similar to the way in which new services are added. The WebLogic Server on which the service was undeployed broadcasts a message indicating that it no longer provides the service. Again, other servers in the cluster that observe the multicast message update their local copies of the JNDI tree to indicate that the service is no longer available on the server that undeployed the object.

Once the client has obtained a replica-aware stub, the server instances in the cluster may continue adding and removing host servers for the clustered objects, as described in Updating the JNDI Tree. As the information in the JNDI tree changes, the client's stub may also be updated. Subsequent RMI requests contain update information as necessary to ensure that the client stub remains up-to-date.

Client Interaction with the Cluster-Wide JNDI Tree

Clients that connect to a WebLogic Server cluster and look up a clustered object obtain a replica-aware stub for the object. This stub contains the list of available server instances that host implementations of the object. The stub also contains the load balancing logic for distributing the load among its host servers.

"Understanding Object Clustering" on page 4-1 provides more details about replica-aware stubs for EJBs and RMI classes.

For a more detailed discussion of how WebLogic JNDI is implemented in a clustered environment and how to make your own objects available to JNDI clients, see "Using WebLogic JNDI in a Clustered Environment" in *Programming WebLogic JNDI*.

Load Balancing of Clustered Services

In order for a cluster to be scalable, it must ensure that each server is fully utilized. The standard technique for accomplishing this is load balancing. The basic idea behind load balancing is that by distributing the load proportionally among all the servers in the cluster, the servers can each run at full capacity. The trick to load balancing is coming up with a technique that is simple yet sufficient. If all servers in the cluster are the same power and offer the same services, it is possible to use a very simple algorithm that requires no knowledge of the servers. If the servers vary in power or in the kind of services they deploy, the algorithm must take into account these differences.

Load Balancing for HTTP Session States

Load balancing for servlet and JSP HTTP session states can be accomplished either by using separate load balancing hardware or by using the built-in load balancing capabilities of a WebLogic proxy plug-in.

For clusters that utilize a bank of Web servers and WebLogic proxy plug-ins, the proxy plug-ins provide only a round-robin algorithm for distributing requests to servlets and JSPs in a cluster. This load balancing method is described below in Round-Robin (Default).

Clusters that utilize a hardware load balancing solution can utilize any load balancing algorithms supported by the hardware. These may include advanced load-based balancing strategies that monitor the utilization of individual machines.

> **Note:**   External load balancers can distribute HTTP traffic, but do not provide load balancing for EJBs and RMI objects. Object-level load balancing requires specific algorithms and services that current external load balancers do not provide. See "Load Balancing for Clustered Objects" on page 2-16 for a discussion of object-level load balancing in WebLogic Server.

Load Balancing for Clustered Objects

WebLogic Server clusters support several algorithms for load balancing clustered objects. The particular algorithm you choose is maintained within the replica-aware stub obtained for the clustered object. Configurable algorithms for load balancing clustered objects are:

- Round-robin

- Weight-based

- Random

Load balancing is not supported for objects running on a client over the IIOP protocol. For more information, see "Clustered Objects and RMI-IIOP Clients" on page 4-2.

Round-Robin (Default)

WebLogic Server uses the round-robin algorithm as the default load balancing strategy for clustered object stubs when no algorithm is specified. Round-robin is the *only* load balancing strategy used by WebLogic proxy plug-ins for HTTP session state clustering.

The round-robin algorithm cycles through a list of WebLogic Server instances in order. For clustered objects, the server list consists of WebLogic Server instances that host the clustered object. For proxy plug-ins, the list consists of all WebLogic Servers that host the clustered servlet or JSP.

The advantages of this algorithm are that it is simple, cheap and very predictable. The primary disadvantage is that there is some chance of convoying. Convoying occurs when one server is significantly slower than the others. Because replica-aware stubs or proxy plug-ins access the servers in the same order, one slow server can cause requests to "synchronize" on the server, then follow other servers in order for future requests.

Weight-Based

The weight-based algorithm applies only to object clustering. The algorithm improves on the round-robin algorithm by taking into account a pre-assigned weight for each server. Each server in the cluster is assigned a weight in the range (1-100) using the Cluster Weight field in the WebLogic Server Administration Console. This is a declaration of what proportion of the load the server will bear relative to other servers. If all servers have either the default weight (100) or the same weight, they will each bear an equal proportion of the load. If one server has weight 50 and all other servers have weight 100, the 50-weight server will bear half as much as any other server. This algorithm makes it possible to apply the advantages of the round-robin algorithm to clusters that are not homogeneous.

If you use the weight-based algorithm, you should spend some time to accurately determine the relative weights to assign to each server instance. Factors that could affect a server's assigned weight include:

■ The processing capacity of the server's hardware in relationship to other servers (for example, the number and performance of CPUs dedicated to WebLogic Server).

■ The number of non-clustered ("pinned") objects each server hosts.

If you change the specified weight of a server and reboot it, the new weighting information is propagated throughout the cluster via the replica-aware stubs. See "Cluster-Wide JNDI Naming Service" on page 2-10 for more information.

Note: In this version of WebLogic Server, weight-based load balancing is not supported for objects that communicate using the RMI/IIOP protocol.

Random

This algorithm applies only to object clustering. The algorithm chooses the next replica at random. This will tend to distribute calls evenly among the replicas. It is only recommended in a clusters where each server has the same power and hosts the same services. The advantages are that it is simple and relatively cheap. The primary disadvantage is that there is a small cost to generating a random number on every request, and there is a slight probability that the load will not be evenly balanced over a small number or runs.

Using Parameter-Based Routing for Clustered Objects

It is also possible to gain finer grain control over load balancing. Any clustered object can be assigned a `CallRouter`. This is a class that is called before each invocation with the parameters of the call. The `CallRouter` is free to examine the parameters and return the name server to which the call should be routed. See Appendix A, "The WebLogic Cluster API," for information about creating custom `CallRouter` classes.

Load Balancing and JDBC Connections

Load balancing of JDBC connection requires the use of a multipool configured for load balancing. Load balancing support is an option you can choose when configuring a multipool.

A load balancing multipool provides the high available behavior described in "Failover and JDBC Connections" on page 2-21, and in addition, balances the load among the connection pools in the multipool. A multipool has an ordered list of connection pools it contains. If you do not configure the multipool for load balancing, it always attempts to obtain a connection from the first connection pool in the list. In a load-balancing multipool, the connection pools it contains are accessed using a round-robin scheme. In each successive client request for a multipool connection, the list is rotated so the first pool tapped cycles around the list.

For instructions to cluster JDBC connections, see "Configure Clustered JDBC" on page 6-17.

Load Balancing and JMS

You can load balance JMS destinations across multiple Managed Servers in the cluster by configuring multiple JMS servers and using targets to assign them to the defined WebLogic Servers. Each JMS server is deployed on exactly one WebLogic Server and handles requests for a set of destinations.

Load balancing is not dynamic. During configuration, the system administrator defines load balancing by specifying targets for JMS servers. For instructions on setting JMS for a cluster, see "Configure JMS" on page 6-19.

Failover Support for Clustered Services

In order for a cluster to provide high availability it must be able to recover from service failures. This section describes how WebLogic Server detect failures in a cluster, and provides an overview of how failover works for replicated HTTP session states and clustered objects.

How WebLogic Server Detects Failures

WebLogic Server instances in a cluster detect failures of their peer server instances by monitoring:

- Socket connections to a peer server

- Regular server "heartbeat" messages

Failure Detection Using IP Sockets

WebLogic Servers monitor the use of IP sockets between peer server instances as an immediate method of detecting failures. If a server connects to one of its peers in a cluster and begins transmitting data over a socket, an unexpected closure of that socket causes the peer server to be marked as "failed," and its associated services are removed from the JNDI naming tree.

The WebLogic Server "Heartbeat"

If clustered server instances do not have opened sockets for peer-to-peer communication, failed servers may also be detected via the WebLogic Server "heartbeat." All server instances in a cluster use multicast to broadcast regular server "heartbeat" messages to other members of the cluster. Each server heartbeat contains data that uniquely identifies the server that sends the message. Servers broadcast their heartbeat messages at regular intervals of 10 seconds. In turn, each server in a cluster monitors the multicast address to ensure that all peer servers' heartbeat messages are being sent.

If a server monitoring the multicast address misses three heartbeats from a peer server (i.e., if it does not receive a heartbeat from the server for 30 seconds or longer), the monitoring server marks the peer server as "failed." It then updates its local JNDI tree, if necessary, to retract the services that were hosted on the failed server.

In this way, servers can detect failures even if they have no sockets open for peer-to-peer communication.

Failover for Clustered Servlets and JSPs

For clusters that utilize Web servers with WebLogic proxy plug-ins, the proxy plug-in handles failover transparently to the client. If a given server fails, the plug-in locates the replicated HTTP session state on a secondary server and redirects the client's request accordingly.

For clusters that use a supported hardware load balancing solution, the load balancing hardware simply redirects client requests to any available server in the WebLogic Server cluster. The cluster itself obtains the replica of the client's HTTP session state from a secondary server in the cluster.

Chapter 3, "Understanding HTTP Session State Replication" describes the failover procedure for replicated HTTP session states in more detail.

Failover for Clustered Objects

For clustered objects, failover is accomplished using the object's replica-aware stub. When a client makes a call through a replica-aware stub to a service that fails, the stub detects the failure and retries the call on another replica.

Failover is not supported for objects running on a client over the IIOP protocol. For more information, see "Clustered Objects and RMI-IIOP Clients" on page 4-2.

Idempotent Objects

With clustered objects, automatic failover generally occurs only in cases where the object is *idempotent*. An object is idempotent if any method can be called multiple times with no different effect than calling the method once. This is always true for methods that have no permanent side effects. Methods that do have side effects have to be written specially with idempotence in mind.

Consider a shopping cart service call `addItem()` that adds an item to a shopping cart. Suppose client C invokes this call on a replica on Server S1. After S1 receives the call, but before it successfully returns to C, S1 crashes. At this point the item has been added to the shopping cart, but the replica-aware stub has received an exception. If the stub were to retry the method on Server S2, the item would be added a second time to the shopping cart. Because of this, replica-aware stubs will not, by default, attempt to retry a method that fails after the request is sent but before it returns. This behavior can be overridden by marking a service idempotent.

Other Failover Exceptions

Even if a clustered object is not idempotent, WebLogic Server performs automatic failover in the case of a `ConnectException` or `MarshalException`. Either of these exceptions indicates that the object could not have been modified, and therefore there is no danger of causing data inconsistency by failing over to another instance.

Failover and JDBC Connections

JDBC is a highly stateful client-DBMS protocol, in which the DBMS connection and transactional state are tied directly to the socket between the DBMS process and the client (driver). For this reason, failover of a connection is not supported. If a WebLogic Server instance dies, any JDBC connections that it managed will die, and the DBMS(s) will roll back any transactions that were under way. Any applications affected will have to restart their current transactions from the beginning. All JDBC objects associated with dead connections will also be defunct. Clustered JDBC eases the reconnection process: the cluster-aware nature of WebLogic data sources in external client applications allow a client to request another connection from them if the server instance that was hosting the previous connection fails.

If you have replicated, synchronized database instances, you can use a JDBC multipool to support database failover. In such an environment, if a client cannot obtain a connection from one connection pool in the multipool because the pool doesn't exist or because database connectivity from the pool is down, WebLogic Server will attempt to obtain a connection from the next connection pool in the list of pools.

Notes: If a client requests a connection for a pool in which all the connections are in use, an exception is generated, and WebLogic Server will not attempt to obtain a connection from another pool. You can address this problem by increasing the number of connections in the connection pool.

Any connection pool assigned to a multipool must be configured to test its connections at reserve time. This is the only way a pool can verify it has a good connection, and the only way a multipool can know when to fail over to the next pool on its list.

For instructions to cluster JDBC connections, see "Configure Clustered JDBC" on page 6-17.

Failover and JMS

Automatic failover is not supported by WebLogic JMS for this release. For information about how to restart or replace a WebLogic Server instance in the event of a system failure, and provide programming considerations for gracefully terminating a JMS application following such an event.performing a manual failover, refer to "Recovering from a WebLogic Server Failure" in *WebLogic Server Administration Guide*.

3 Understanding HTTP Session State Replication

The following section describe HTTP session state replication including:

- Overview

- Requirements for HTTP Session State Replication

- Using Replication Groups

- Accessing Clustered Servlets and JSPs Using a Proxy

- Accessing Clustered Servlets and JSPs with Load Balancing Hardware

- Delayed Replication after a Failure

Overview

To support automatic failover for servlet and JSP HTTP session states, WebLogic Server replicates the session state object in memory. This process creates a primary session state, which resides on the WebLogic Server to which the client first connects, and a secondary replica of the session state on another WebLogic Server instance in the cluster. The replica is always kept up-to-date so that it may be used if the server

that hosts the servlet fails. The process of copying a state from one instance to another is called *in-memory replication*. For instructions fon setting up in-memory replication see "Configure In-Memory HTTP Replication" on page 6-20

Note: WebLogic Server also provides the ability to maintain the HTTP session state of a servlet or JSP using file-based or JDBC-based persistence. For more information on these persistence mechanisms, see "Making Sessions Persistent" in *Programming WebLogic HTTP Servlets*.

Requirements for HTTP Session State Replication

To utilize in-memory replication for HTTP session states, you must access the WebLogic Server cluster using either:

- Load balancing hardware, or

- A collection of Web servers with WebLogic proxy plug-ins (configured identically)

Proxy Requirements

The WebLogic proxy plug-ins maintain a list of WebLogic Server instances that host a clustered servlet or JSP, and forward HTTP requests to those instances using a simple round-robin strategy. The proxy also provides the logic required to locate the replica of a client's HTTP session state if a WebLogic Server instance should fail.

Supported Web server and proxy software includes:

- WebLogic Server with the `HttpClusterServlet`

- Netscape Enterprise Server with the Netscape (proxy) plug-in

- Apache with the Apache Server (proxy) plug-in

- Microsoft Internet Information Server with the Microsoft-IIS (proxy) plug-in

Load Balancer Requirements

If you choose to use load balancing hardware instead of a proxy plug-in, you must use hardware that supports:

■ A compatible passive or active cookie persistence mechanism, and

■ SSL persistence

Passive cookie persistence enables WebLogic Server to write cookies (containing information about the location of replicated HTTP session states) through the load balancer to the client. The load balancer, in turn, interprets a string identifier in the client's cookie to maintain the relationship between the client and the primary WebLogic Server hosting the HTTP session state.

Certain active cookie persistence mechanisms can also be used with WebLogic Server clusters, provided the load balancer *does not* modify the WebLogic Server cookie.

When SSL persistence is used, the load balancing product performs all encryption and decryption of data between clients and the WebLogic Server cluster. The load balancer then uses the plain text cookie that WebLogic Server inserts on the client to maintain an association between the client and a particular server in the cluster.

See Configure Load Balancing Hardware (Optional) for details on setting up supported load balancing solutions with WebLogic Server. See Configuring BIG-IP™ Hardware with Clusters for specific information on configuring BIG-IP load balancers.

Session Programming Requirements

When developing servlets or JSPs that you will deploy in a clustered environment, keep in mind the following requirements.

Session Data Must Be Serializable

In order to support in-memory replication for HTTP session states, all servlet and JSP session data *must be serializable*. If the servlet or JSP uses a combination of serializable and non-serializable objects, WebLogic Server does not replicate the session state of the non-serializable objects.

Use setAttribute to Change Session State

In an HTTP servlet that implements `javax.servlet.http.HttpSession`, use `HttpSession.setAttribute` (which replaces the deprecated `putValue`) to change attributes in a session object. If you set attributes in a session object with `setAttribute`, the object and its attributes are replicated in a cluster using in-memory replication. If you use other set methods to change objects within a session, WebLogic Server does not replicate those changes.

Likewise, use `removeAttribute` (which, in turn, replaces the deprecated `removeValue`) to remove an attribute from a session object.

Note: Use of the deprecated `putValue` and `removeValue` methods will also cause session attributes to be replicated.

Consider Serialization Overhead for Session Objects

Serializing session data introduces some overhead for replicating the session state. The overhead increases as the size of serialized objects grows. If you plan to create very large objects in the session, first test the performance of your servlets to ensure that performance is acceptable.

Applications Using Frames Must Coordinate Session Access

If you are designing a Web application that utilizes multiple frames, keep in mind that there is no synchronization of requests made by frames in a given frameset. For example, it is possible for multiple frames in a frameset to create multiple sessions on behalf of the client application, even though the client should logically create only a single session.

In a clustered environment, poor coordination of frame requests can cause unexpected application behavior. For example, multiple frame requests can "reset" the application's association with a clustered instance, because the proxy plug-in treats each request independently. It is also possible for an application to corrupt session data by modifying the same session attribute via multiple frames in a frameset.

To avoid unexpected application behavior, always use careful planning when accessing session data with frames. You can apply one of the following general rules to avoid common problems:

- In a given frameset, ensure that only one frame creates and modifies session data.

- Always create the session in a frame of the first frameset your application uses (for example, create the session in the first HTML page that is visited). After the session has been created, access the session data only in framesets other than the first frameset.

Using Replication Groups

By default, WebLogic Server attempts to create session state replicas on a different machine than the one hosting the primary session state. WebLogic Server enables you to further control where secondary states are placed using *replication groups*. A replication group is simply a preferred list of clustered instances to be used for storing session state replicas.

Using the WebLogic Server administration console, you can define unique machine names that will host individual server instances. These machine names can be associated with new WebLogic Server instances to identify where the servers reside in your system. Machine names are generally used to indicate servers that run on multihomed machines. For example, you would assign the same machine name to all server instances that run on the same multihomed machine, or the same server hardware.

If you do not use a multihomed machine, or you do not run multiple WebLogic Server instances on a single piece of hardware, you do not need to specify WebLogic Server machine names. Servers without an associated machine name are automatically treated as though they reside on separate, physical hardware. See Define Machine Names (Optional) for detailed instructions on setting machine names.

When you configure a clustered server instance, you can also assign the server membership within a replication group, as well as the *preferred secondary replication group* to be considered for hosting replicas of the primary HTTP session states created on the server.

When a client attaches to a server in the cluster and creates a primary session state, the server hosting the primary state ranks other servers in the cluster to determine which server should host the secondary. Server ranks are assigned using a combination of the

server's location (whether or not it resides on the same machine as the primary server) and its participation in the primary server's preferred replication group. The following table shows the relative ranking of servers in a cluster.

Server Rank	Server Resides on a Different Machine?	Server Is a Member of Preferred Replication Group?
1	Yes	Yes
2	No	Yes
3	Yes	No
4	No	No

Using the ranking rules above, the primary WebLogic Server ranks other members of the cluster and chooses the highest-ranked server to host the secondary session state. For example, the following figure shows replication groups configured to account for distinct geographic locations.
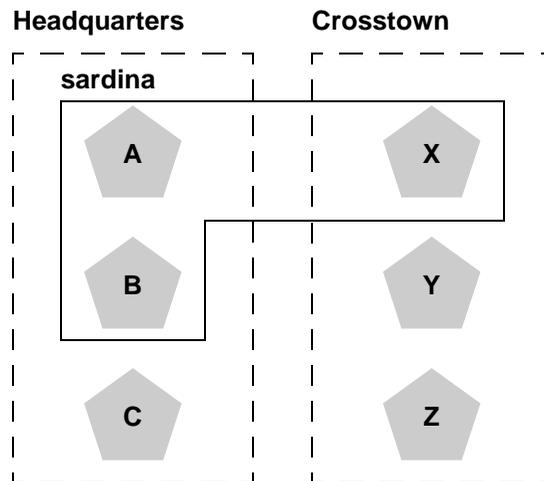
In this example, Servers A, B, and C are members of the replication group "Headquarters" and use the preferred secondary replication group "Crosstown." Conversely, Servers X, Y, and Z are members of the "Crosstown" group and use the preferred secondary replication group "Headquarters." Servers A, B, and X reside on the same machine, "sardina."

If a client connects to Server A and creates an HTTP session state, servers Y and Z are most likely to host the replica of this state, since they reside on separate machines and are members of Server A's preferred secondary group. Server X holds the next-highest ranking because it is also a member of the preferred replication group (even though it resides on the same machine as the primary.) Server C holds the third-highest ranking since it resides on a separate machine but is not a member of the preferred secondary group. Server B holds the lowest ranking, because it resides on the same machine as Server A (and could potentially fail along with A if there is a hardware failure) and it is not a member of the preferred secondary group.

To define machine names for clustered WebLogic Server instances, use the instructions in Define Machine Names (Optional). To configure a server's membership in a replication group, or to assign a server's preferred secondary replication group, use the instructions in Configure Replication Groups (Optional).

Accessing Clustered Servlets and JSPs Using a Proxy

This section describes how WebLogic Server accesses clustered servlets and JSPs in a configuration that includes a WebLogic proxy plug-in.

The WebLogic proxy plug-in maintains a list of WebLogic Server instances that host a clustered servlet or JSP, and forwards HTTP requests to those instances using a round-robin strategy. The plug-in also provides the logic necessary to locate the replica of a client's HTTP session state if a WebLogic Server instance should fail.

WebLogic Server supports the following Web servers and associated proxy plug-ins:

- WebLogic Server with the HttpClusterServlet

- Netscape Enterprise Server with the Netscape (proxy) plug-in

- Apache with the Apache Server (proxy) plug-in

- Microsoft Internet Information Server with the Microsoft-IIS (proxy) plug-in

For instructions on setting up a proxy, see *WebLogic Server Administration Guide*.

The following figure depicts a client accessing a servlet hosted in a two-tier cluster architecture. This example uses a single WebLogic Server to serve static HTTP requests only; all servlet requests are forwarded to the WebLogic Server via the `HttpClusterServlet`.
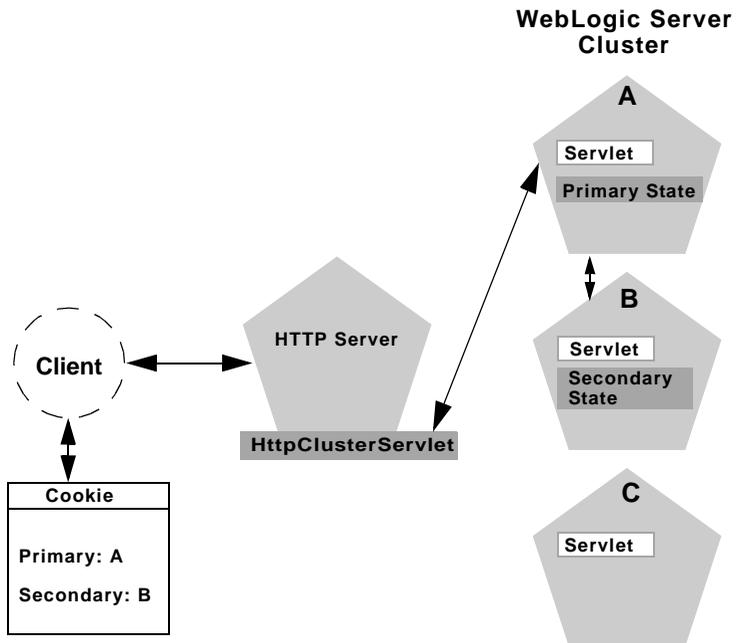
Note: The discussion that follows also applies if you use a third-party Web server and WebLogic proxy plug-in, rather than WebLogic Server and the `HttpClusterServlet`.

When the HTTP client requests the servlet, the `HttpClusterServlet` proxies the request to the WebLogic Server cluster. The `HttpClusterServlet` maintains the list of all servers in the cluster, as well as the load balancing logic to use when accessing

the cluster. In the above example, the `HttpClusterServlet` routes the client request to the servlet hosted on WebLogic Server A. WebLogic Server A becomes the primary server hosting the client's servlet session.

To provide failover services for the servlet, the primary server replicates the client's servlet session state to a secondary WebLogic Server in the cluster. This ensures that a replica of the session state exists even if the primary server fails (for example, due to a network failure). In the example above, Server B is selected as the secondary.

The servlet page is returned to the client through the `HttpClusterServlet`, and the client browser is instructed to write a cookie that lists the primary and secondary locations of the servlet session state. If the client browser does not support cookies, WebLogic Server can use URL rewriting instead.

Using URL Rewriting to Track Session Replicas

In its default configuration, WebLogic Server uses client-side cookies to keep track of the primary and secondary server that host the client's servlet session state. If client browsers have disabled cookie usage, WebLogic Server can also keep track of primary and secondary servers using URL rewriting. With URL rewriting, both locations of the client session state are embedded into the URLs passed between the client and proxy server. To support this feature, you must ensure that URL rewriting is enabled on the WebLogic Server cluster. See Configuring Session Cookies for more information.

Proxy Failover Procedure

Should the primary server fail, the `HttpClusterServlet` uses the client's cookie information to determine the location of the secondary WebLogic Server that hosts the replica of the session state. The `HttpClusterServlet` automatically redirects the client's next HTTP request to the secondary server, and failover is transparent to the client.

After the failure, WebLogic Server B becomes the primary server hosting the servlet session state, and a new secondary is created (Server C in the example above). In the HTTP response, the proxy updates the client's cookie to reflect the new primary and secondary servers, to account for the possibility of subsequent failovers.

In a two-server cluster, the client would transparently fail over to the server hosting the secondary session state. However, replication of the client's session state would not continue unless another WebLogic Server became available and joined the cluster. For example, if the original primary server was restarted or reconnected to the network, it would be used to host the secondary session state.

Accessing Clustered Servlets and JSPs with Load Balancing Hardware

This section describes how WebLogic Server accesses clustered servlets and JSPs in a configuration that includes load balancing hardware. For instructions on setting up an external load balancer, see "Configure Load Balancing Hardware (Optional)" on page 6-13.

To support direct client access via load balancing hardware, the WebLogic Server replication system enables clients to use secondary session states regardless of the server to which the client fails over. WebLogic Server version 6.1 continues to use client-side cookies or URL rewriting to record primary and secondary server locations. However, this information is used only as a history of the servlet session state location; when accessing a cluster via load balancing hardware, clients do not use the cookie information to actively locate a server after a failure. The following steps describe the connection and failover procedure when using HTTP session state replications with load balancing hardware.
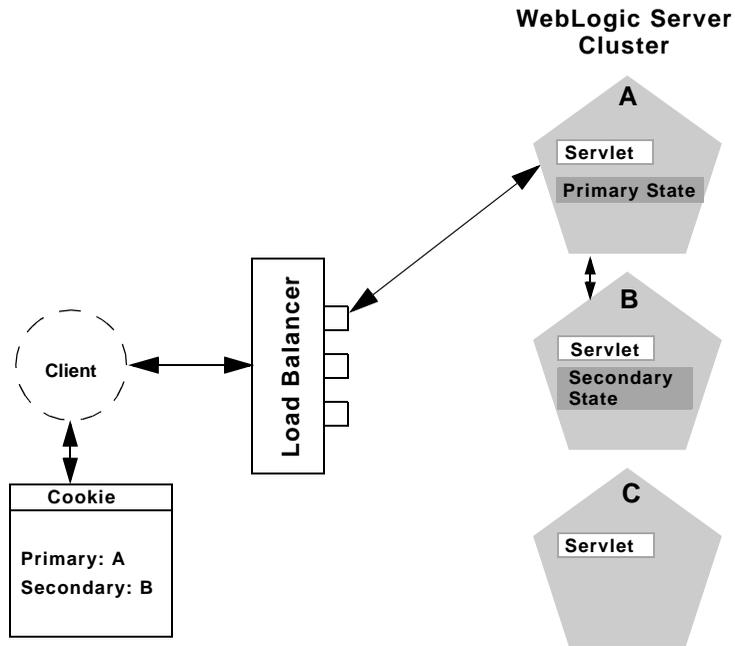
When the client of a Web application requests a servlet using a public IP address:

1. The client's connection request is routed to a WebLogic Server cluster via load balancing hardware. The load balancer uses its configured policies and directs the client request to WebLogic Server A.

2. WebLogic Server A acts as the primary host of the clients servlet session state. It uses the ranking system described in Using Replication Groups to select a server to host the replica of the session state. In the example above, WebLogic Server B is selected to host the replica.

3. The client is instructed to record the location of WebLogic Servers A and B in a local cookie. If the client does not allow cookies, the record of the primary and secondary servers can be recorded in the URL returned to the client via URL rewriting.

Note: You must enable WebLogic Server URL rewriting capabilities to support clients that disallow cookies. See Using URL Rewriting for more information.

4. As the client makes further requests to the cluster, the load balancer uses an identifier in the client-side cookie to ensure that those requests continue to go to WebLogic Server A (rather than being load-balanced to another server in the cluster). This ensures that the client remains associated with the server hosting the primary session object for the life of the session.

Failover with Load Balancing Hardware

Should Server A fail during the course of the client's session, the client's next connection request to Server A also fails.
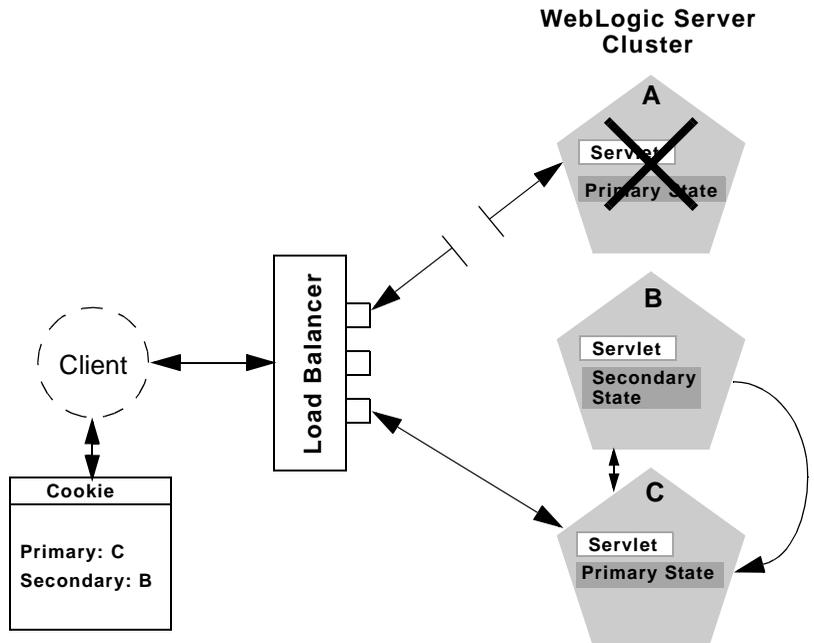
In response to the connection failure:

1. The load balancing hardware uses its configured policies to direct the request to an available WebLogic Server in the cluster. In the above example, assume that the load balancer routes the client's request to WebLogic Server C after WebLogic Server A fails.

2. When the client connects to WebLogic Server C, the server uses the information in the client's cookie (or the information in the HTTP request if URL rewriting is used) to acquire the session state replica on WebLogic Server B. The failover process remains completely transparent to the client.

3. WebLogic Server C becomes the new host for the client's primary session state, and WebLogic Server B continues to host the session state replica. This new information about the primary and secondary host is again updated in the client's cookie, or via URL rewriting.

Delayed Replication after a Failure

The proxy and load balancing examples both depict an important quality of WebLogic Server session state replication: After a server that hosts a session state fails, replication of that server's session states is performed in a delayed fashion, in response to clients attempting to access the session state on another server in the cluster.

For example, in the sequence described in "Failover with Load Balancing Hardware" on page 3-12, the failure of Server A does not immediately cause the cluster to generate a new copy of the session state on Server C. Instead, the session state is copied to Server C only after the client connected to Server C and attempted to use the session state. (Had the client reconnected to Server B instead, the cluster would also replicate the session state to Server C to ensure that at least one copy of the active session state existed in the cluster).

Delaying replication after a failure provides two important benefits:

■ It prevents overloading the cluster with replication requests immediately after a server failure.

■ It prevents replicating session states that might become otherwise remain unused by clients.

The delayed replication behavior does pose a small risk of replication failure in the special case of a cluster having only two WebLogic Server instances. The following sequence shows a potential replication failure with a cluster having two nodes, Servers A and B:

1. A client accesses Server A and creates a new session state; the cluster replicates the session state to Server B.

2. Server A fails, leaving Server B hosting the session state replica.

3. The client again establishes a connection to the cluster, accessing Server B and the replicated session state. Because Server B is the only instance currently in the cluster, the local session state is marked as a primary session state, and no replica can be created.

4. Server A is rebooted, but Server A does not host a replica of the session state, because of the delayed replication behavior.

   In the current state, a failure of Server B causes the client to lose the session state.

To avoid the possibility of losing a session state in this manner, use a cluster having three or more WebLogic Server instances.

4 Understanding Object Clustering

The following sections describe object clustering including:

- Overview

- Replica-Aware Stubs

- Clustered EJBs

- Clustered RMI Objects

- Stateful Session Bean Replication

- Optimization for Collocated Objects

- Object Deployment Requirements

Overview

If an object is clustered, instances of the object are deployed on all WebLogic Servers in the cluster. The client has a choice about which instance of the object to call. Each instance of the object is referred to as a *replica*.

The key technology that underpins clustered objects in WebLogic Server is the *replica-aware stub*. When you compile an EJB that supports clustering (as defined in its deployment descriptor), `ejbc` passes the EJB's interfaces through the `rmic`

compiler to generate replica-aware stubs for the bean. For RMI objects, you generate replica-aware stubs explicitly using command-line options to rmic, as described in WebLogic RMI Compiler.

Replica-Aware Stubs

A replica-aware stub appears to the caller as a normal RMI stub. Instead of representing a single object, however, the stub represents a collection of replicas. The replica-aware stub contains the logic required to locate an EJB or RMI class on any WebLogic Server instance on which the object is deployed. When you deploy a cluster-aware EJB or RMI object, its implementation is bound into the JNDI tree. As described in Cluster-Wide JNDI Naming Service, clustered WebLogic Server instances have the capability to update the JNDI tree to list all servers on which the object is available. When a client accesses a clustered object, the implementation is replaced by a replica-aware stub, which is sent to the client.

The stub contains the load balancing algorithm (or the call routing class) used to load balance method calls to the object. On each call, the stub can employ its load algorithm to choose which replica to call. This provides load balancing across the cluster in a way that is transparent to the caller. If a failure occurs during the call, the stub intercepts the exception and retries the call on another replica. This provides a failover that is also transparent to the caller.

Clustered Objects and RMI-IIOP Clients

Clustering support for RMI objects that run over the IIOP protocol is limited to server-side objects. Clients that use the JDK orb do not have access to WebLogic classes and cannot take advantage of WebLogic-specific features, including load balancing and failover. Load balancing and failover is suppported for clustered objects running over IIOP only if they run within the WebLogic Server runtime environment. Client-side RMI objects must run over the T3 protocol to support load balancing and failover.

Clustered EJBs

EJBs differ from plain RMI objects in that each EJB can potentially generate two different replica-aware stubs: one for the EJBHome interface and one for the EJBObject interface. This means that EJBs can potentially realize the benefits of load balancing and failover on two levels:

- When a client looks up an EJB object using the EJBHome stub

- When a client makes method calls against the EJB using the EJBObject stub

The following sections provide an overview of the capabilities of different EJBs. See EJBs in WebLogic Server Clusters for a detailed explanation of the clustering behavior for different EJB types.

EJB Home Stubs

All bean homes can be clustered. When a bean is deployed on a server, its home is bound into the cluster-wide naming service. Because homes can be clustered, each server can bind an instance of the home under the same name. When a client looks up this home, it gets a replica-aware stub that has a reference to the home on each server that deployed the bean. When create() or find() is called, the replica-aware stub routes the call to one of the replicas. The home replica receives the find() results or creates an instance of the bean on this server.

Stateless EJBs

When a home creates a stateless bean, it returns a replica-aware EJBObject stub that can route to any server on which the bean is deployed. Because a stateless bean holds no state on behalf of the client, the stub is free to route any call to any server that hosts the bean. Also, because the bean is clustered, the stub can automatically fail over in the event of a failure. The stub does not automatically treat the bean as idempotent, so it will not recover automatically from all failures. If the bean has been written with idempotent methods, this can be noted in the deployment descriptor and automatic failover will be enabled in all cases.

Stateful EJBs

As with all EJBs, clustered stateful session EJBs utilize a replica-aware `EJBHome` stub. If you use stateful session EJB replication, the EJB also utilizes a replica-aware `EJBObject` stub that maintains the location of the EJB's primary and secondary states. The state of the EJB is maintained using a replication scheme similar to that used for HTTP session states. See Stateful Session Bean Replication for more information.

Entity EJBs

There are two types of entity beans to consider: read-write entities and read-only entities.

When a home finds or creates a read-write entity bean, it obtains an instance on the local server and returns a stub pinned to that server. Load balancing and failover occur only at the home level. Because it is possible for multiple instances of the entity bean to exist in the cluster, each instance must read from the database before each transaction and write on each commit.

When a home finds or creates a read-only entity bean, it returns a replica-aware stub. This stub load balances on every call but does not automatically fail over in the event of a recoverable call failure. Read-only beans are also cached on every server to avoid database reads.

For more information about using EJBs in a cluster, please read The WebLogic Server EJB Container.

Failover for Entity Beans and EJB Handles

Failover for entity beans and EJB handles requires that the cluster address be specified as a DNS name that maps to *all* server instances in the cluster and *only* server instances in the cluster. The cluster DNS name should not map to a server instance that is not a member of the cluster.

Clustered RMI Objects

WebLogic RMI provides special extensions for building clustered remote objects. These are the extensions used to build the replica-aware stubs described in the EJB section. For more information about using RMI in WebLogic Server Clusters, see Using WebLogic RMI.

Stateful Session Bean Replication

WebLogic Server replicates the state of stateful session EJBs similar to the way in which it replicates HTTP session states. When a client creates the `EJBObject` stub, the point-of-contact WebLogic Server instance automatically selects a secondary server instance to host the replicated state of the EJB. Secondary server instances are selected using the same rules defined in Understanding HTTP Session State Replication. For example, you can define a collection of WebLogic Server instances to act as a replication group for hosting replicating stateful session EJB data.

The client receives a replica-aware stub that lists the location of the primary and secondary servers in the cluster that host the EJB's state. The following figure shows a client accessing a clustered stateful session EJB.
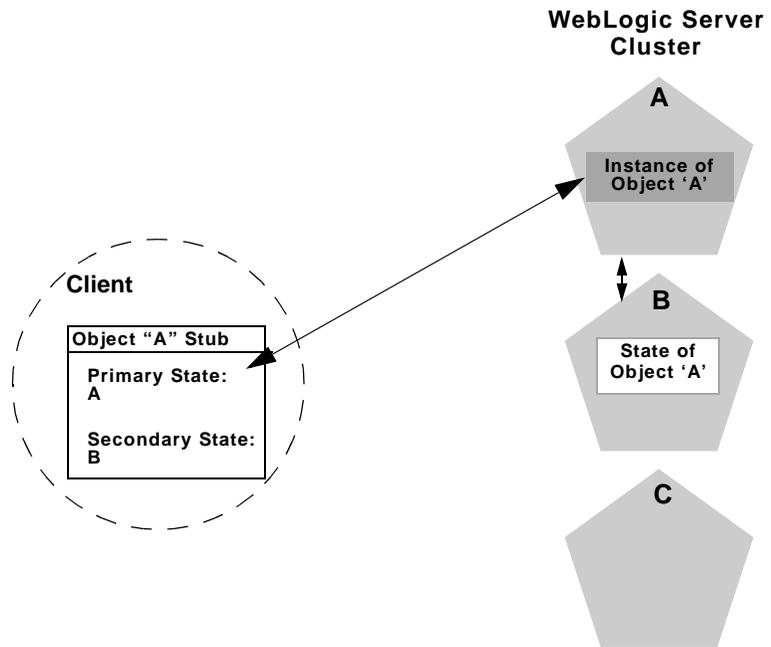
The primary server hosts the actual instance of the EJB that the client interacts with. The secondary server hosts only the replicated state of the EJB, which consumes a small amount of memory. The secondary sever does not create an actual instance of the EJB unless a failover occurs. This ensures minimal resource usage on the secondary server; you do not need to configure additional EJB resources to account for replicated EJB states.

Replicating EJB State Changes

As the client makes changes to the state of the EJB, state differences are replicated to the secondary server instance. For EJBs that are involved in a transaction, replication occurs immediately after the transaction commits. For EJBs that are not involved in a transaction, replication occurs after each method invocation.

In both cases, only the actual changes to the EJB's state are replicated to the secondary server. This ensures that there is minimal overhead associated with the replication process.

Note: The actual state of a stateful EJB is non-transactional, as described in the EJB specification. Although it is unlikely, there is a possibility that the current state of the EJB can be lost. For example, if a client commits a transaction involving the EJB and there is a failure of the primary server *before* the state change is replicated, the client will fail over to the previously-stored state of the EJB.

If it is critical to preserve the state of your EJB in all possible failover scenarios, use an entity EJB rather than a stateful session EJB.

Failover for Stateful Session EJBs

Should the primary server fail, the client's EJB stub automatically redirects further requests to the secondary WebLogic Server instance. At this point, the secondary server creates a new EJB instance using the replicated state data, and processing continues on the secondary server.

After a failover, WebLogic Server chooses a new secondary server to replicate EJB session states (if another server is available in the cluster). The location of the new primary and secondary server instances is automatically updated in the client's replica-aware stub on the next method invocation, as shown below.
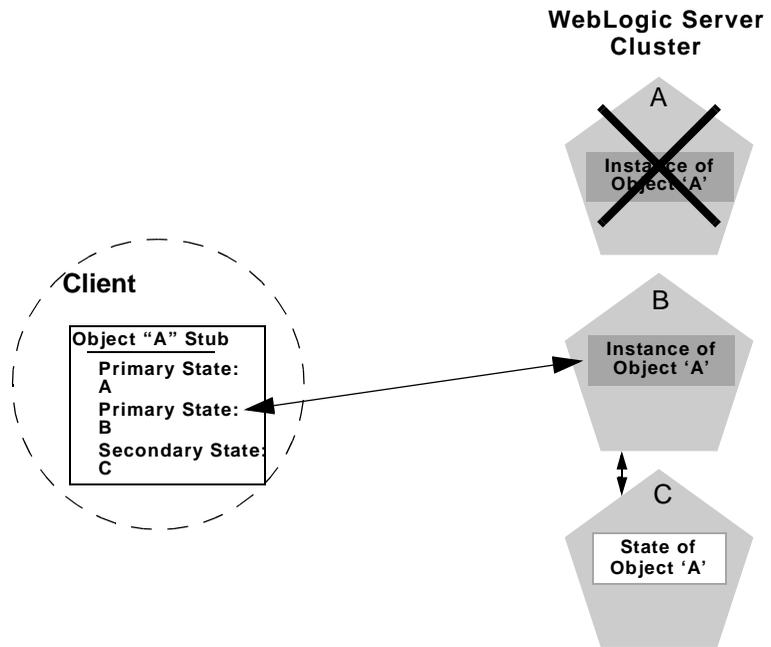
Optimization for Collocated Objects

Although a replica-aware stub contains the load-balancing logic for a clustered object, WebLogic Server does not always perform load balancing for an object's method calls. In most cases, it is more efficient to use a replica that is collocated with the stub itself, rather than using an replica that resides on a remote server. The figure below details this situation.

**WebLogic Server
Cluster**

Client

Servlet Objects
Stub A
B
C

Servlet Objects
A
B
C

Database

Servlet Objects
A
B
C
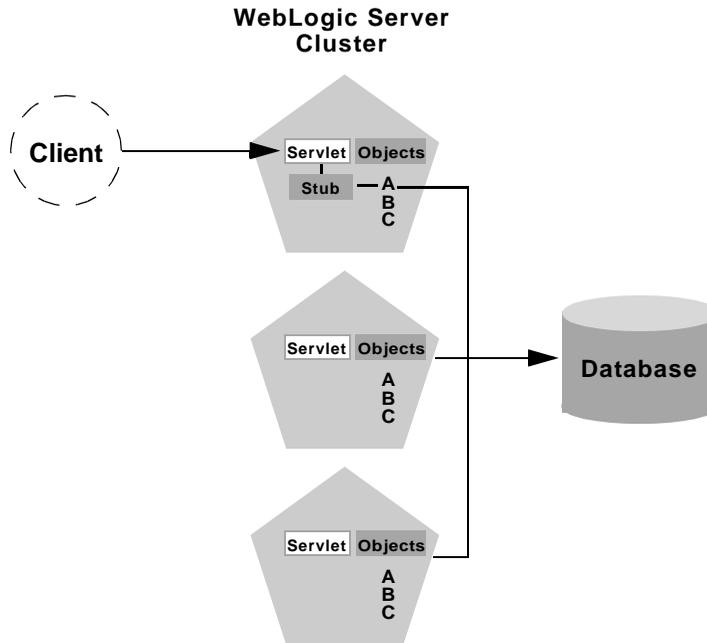
In the above example, a client connects to a servlet hosted by the first WebLogic Server instance in the cluster. In response to client activity, the servlet obtains a replica-aware stub for Object A. Because a replica of Object A is also available on the same server, the object is said to be collocated with the client's stub.

WebLogic Server always uses the local, collocated copy of Object A, rather than distributing the client's calls to other replicas of Object A in the cluster. It is more efficient to use the local copy, because doing so avoids the network overhead of establishing peer connections to other servers in the cluster.

This optimization is often overlooked when planning WebLogic Server clusters. The collocation optimization is also frequently confusing for administrators or developers who expect or require load balancing on each method call. In single-cluster Web architectures, this optimization *overrides* any load balancing logic inherent in the replica-aware stub.

If you require load balancing on each method call to a clustered object, see Planning WebLogic Server Clusters for information about how to plan your WebLogic Server cluster accordingly.

Transactional Collocation

As an extension to the basic collocation strategy, WebLogic Server also attempts to collocate clustered objects that are enlisted as part of the same transaction. When a client creates a UserTransaction object, WebLogic Server attempts to use object replicas that are collocated with the transaction. This optimization is depicted in the figure below.
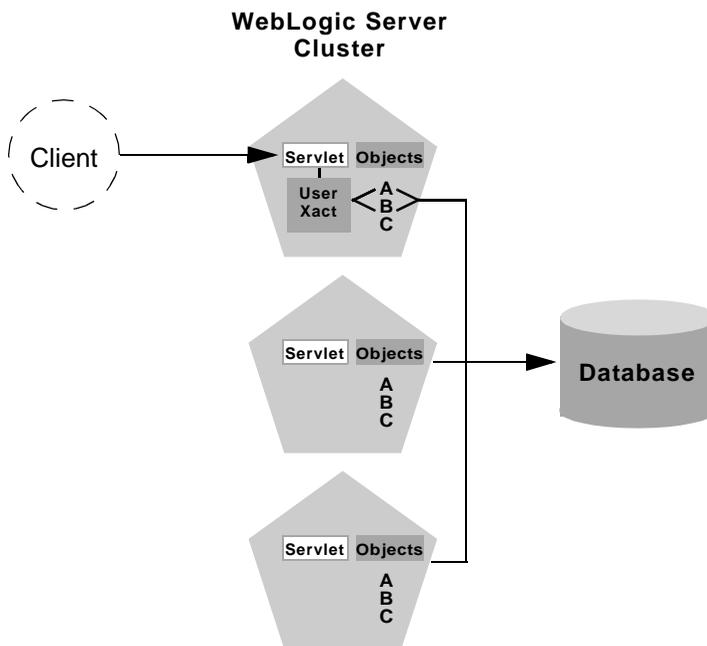
In this example, a client attaches to the first WebLogic Server instance in the cluster and obtains a UserTransaction object. After beginning a new transaction, the client looks up Objects A and B to do the work of the transaction. In this situation WebLogic

Server always attempts to use replicas of A and B that reside on the same server as the `UserTransaction` object, regardless of the load balancing strategies in the stubs for A and B.

This transactional collocation strategy is even more important than the basic optimization described in Optimization for Collocated Objects. If remote replicas of A and B were used, added network overhead would be incurred *for the duration of the transaction*, because the peer connections for A and B would be locked until the transaction committed. Furthermore, WebLogic Server would need to employ a multi-tiered JDBC connection to commit the transaction, incurring even further network overhead.

By collocating clustered objects in a transactional context, WebLogic Server reduces the network load for accessing the individual objects. The server also can make use of a single-tiered JDBC connection, rather than a multi-tiered connection, to do the work of the transaction.

Object Deployment Requirements

If you are programming EJBs to be used in a WebLogic Server cluster, read the instructions in this chapter and in The WebLogic Server EJB Container to understand the capabilities of different EJB types in a cluster. Then ensure that you enable clustering in the EJB's deployment descriptor. weblogic-ejb-jar.xml Deployment Descriptors describes the XML deployment elements relevant for clustering.

If you are developing either EJBs or custom RMI objects, also refer to Using WebLogic JNDI in a Clustered Enviroment to understand the implications of binding clustered objects in the JNDI tree.

5 Planning WebLogic Server Clusters

The following sections describe common issues to consider before deploying one or more WebLogic Server clusters including:

- Overview

- Recommended Basic Cluster

- Planning by Dividing Application Tiers

- Recommended Multi-Tier Architecture

- Recommended Proxy Architectures

- Administration Server Considerations

- Security Options for Cluster Architectures

- Firewall Considerations for Clusters

Overview

You should read this document along with the Introduction to WebLogic Server Clustering and Administering WebLogic Clusters sections to become familiar with how WebLogic Server clusters operate.

This section also presents recommended cluster architectures for WebLogic Server. You should examine each recommended architecture to determine which configuration best meets the needs of your Web application.

Capacity Planning

This section focuses on planning the network topology of your clustered system. It describes how to organize one or more WebLogic Server clusters in relation to load balancers, firewalls, and Web servers, to fully utilize load balancing and failover features for your Web application. Although this kind of planning directly influences the capacity of your cluster system, this section does not focus on traditional capacity planning topics. After determining the layout of your cluster system, you should perform rigorous testing using software such as LoadRunner from Mercury Interactive to simulate heavy client usage. By testing your system under heavy loads, you can determine where you may need to add servers or server hardware to support real-world client loads.

WebLogic Servers on Multi-CPU machines

BEA WebLogic Server has no built-in limit for the number of server instances that can reside in a cluster. Large, multi-processor servers such as Sun Microsystems, Inc. Sun Enterprise 10000, therefore, can host very large clusters or multiple clusters.

In most cases, WebLogic Server clusters scale best when deployed with one WebLogic Server instance for every two CPUs. However, as with all capacity planning, you should test the actual deployment with your target web applications to determine the optimal number and distribution of server instances. See Performance Considerations for Multi-CPU Machines for additional information.

Definition of Terms

This section uses the following terms to describe the parts of a clustered system.

Web Application "Tiers"

A Web application is divided into several "tiers" that describe the logical services the application provides. Keep in mind that not all Web applications are alike, and therefore your application may not utilize all of the tiers described below. Also keep in mind that the tiers represent logical divisions of an application's services, and not *necessarily* physical divisions between hardware or software components. In some cases, a single machine running a single WebLogic Server instance can provide all of the tiers described below.

Web Tier

The *web tier* provides static content (for example, simple HTML pages) to clients of a Web application. The web tier is generally the first point of contact between external clients and the Web application. A simple Web application may have a web tier that consists of one or more machines running WebLogic Express, Apache, Netscape Enterprise Server, or Microsoft Internet Information Server.

Presentation Tier

The *presentation tier* provides dynamic content (for example, servlets or Java Server Pages) to clients of a Web application. A cluster of WebLogic Server instances that hosts servlets and/or JSPs comprises the presentation tier of a web application. If the cluster also serves static HTML pages for your application, it encompasses both the web tier and the presentation tier.

Object Tier

The *object tier* provides Java objects (for example, Enterprise JavaBeans or RMI classes) and their associated business logic to a Web application. A WebLogic Server cluster that hosts EJBs provides an object tier.

De-Militarized Zone (DMZ)

The *De-Militarized Zone (DMZ)* is a logical collection of hardware and services that is made available to outside, untrusted sources. In most Web applications, a bank of Web servers resides in the DMZ to allow browser-based clients access to static HTML content.

The DMZ may provide security against outside attacks to hardware and software. However, because the DMZ is available to untrusted sources, it is less secure than an internal system. For example, internal systems may be protected by a firewall that denies all outside access. The DMZ may be protected by a firewall that *hides* access to individual machines, applications, or port numbers, but it still permits access to those services from untrusted clients.

Load Balancer

In this document, the term *load balancer* describes any technology that distributes client connection requests to one or more distinct IP addresses. For example, a simple Web application may use the DNS round-robin algorithm as a load balancer. Larger applications generally use hardware-based load balancing solutions, which can also provide firewall-like security capabilities.

Load balancers provide the capability to associate a client connection with a particular server in the cluster, which is required when using in-memory replication for client session information. With certain load balancing products, you must configure the cookie persistence mechanism to avoid overwriting the WebLogic Server cookie which tracks primary and secondary servers used for in-memory replication. See Configure Load Balancing Hardware (Optional) for more information.

Proxy Plug-In

A *proxy plug-in* is a WebLogic Server extension to Apache, Netscape Enterprise Server, or Microsoft Internet Information Server that accesses clustered servlets provided by a WebLogic Server cluster. The proxy plug-in contains the load balancing logic for accessing servlets and JSPs in a WebLogic Server cluster. Proxy plug-ins also contain the logic for accessing the replica of a client's session state if the primary WebLogic Server hosting their session state fails.

Recommended Basic Cluster

The basic recommended cluster architecture combines all web application tiers and places the related services (static HTTP, presentation logic, and objects) into a single cluster of WebLogic Server instances. This architecture is shown in the figure below.

**WebLogic Server
Cluster
("combined-tier")**
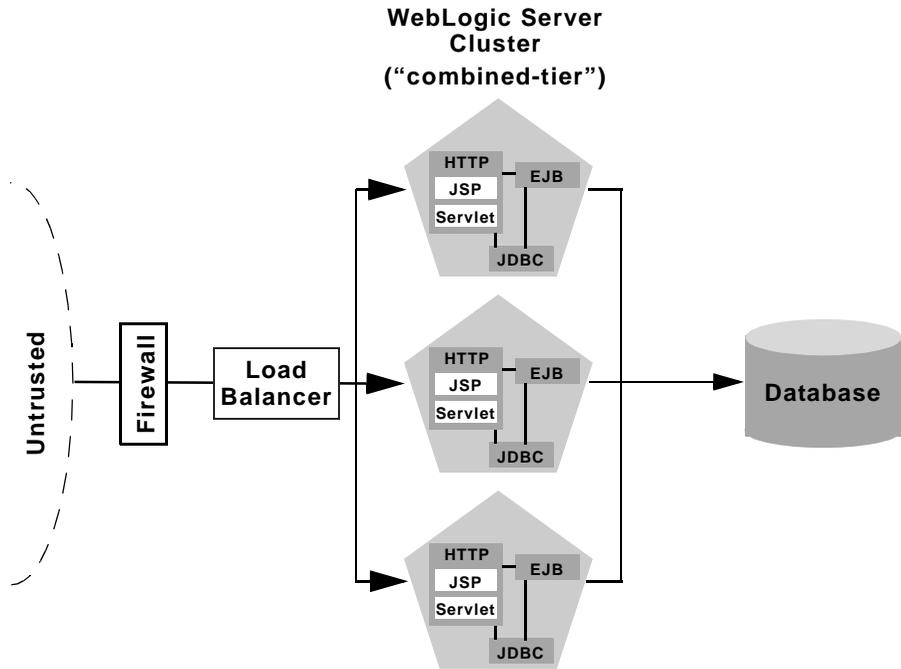
The basic architecture has several strengths:

- **Easy administration**: Because a single cluster hosts static HTTP pages, servlets, and EJBs, you can configure the entire Web application and deploy/undeploy objects using the administration console. You do not need to maintain a separate bank of Web servers (and configure WebLogic Server proxy plug-ins) to benefit from clustered servlets.

- **Flexible load balancing**: Using load balancing hardware directly in front of the WebLogic Server cluster enables you to use advanced load balancing policies for accessing both HTML and servlet content. For example, you may configure your load balancer to detect current server loads and direct client requests appropriately.

- **Robust security**: Placing a firewall in front of your load balancing hardware enables you to set up a De-Militarized Zone (DMZ) for your web application using minimal firewall policies.

- **Optimal performance**: The combined tier architecture offers the best performance for applications in which most or all of the servlets or JSPs in the presentation tier typically access objects in the object tier, such as EJBs or JDBC objects

Note: When using a third-party load balancer with in-memory session replication, you must ensure that the load balancer maintains a client's connection to the WebLogic Server instance that hosts its primary session state (the point-of-contact server). See Configure Load Balancing Hardware (Optional) for more information.

Planning by Dividing Application Tiers

The basic cluster architecture uses a single cluster of WebLogic Server instances to provide all tiers of the web application: web tier, presentation tier, and object tier. In this "combined-tier" cluster, untrusted connections (HTTP and Java clients) have a single interface to the WebLogic Server cluster via load balancing hardware. Although the basic architecture is simplified, it meets the needs of many potential Web applications.

However, two key features of a clustered Web application—load balancing and failover capabilities—can be introduced only at the interfaces between Web application tiers. When those tiers are combined on a single hardware/software platform, as in the basic cluster architecture, the opportunities for introducing load balancing and failover capabilities to your system are reduced.

Because most load balancing and failover occurs between clients and the cluster itself, the basic cluster architecture meets the clustering needs of most Web applications. However, combined-tier clusters provide no opportunity for load balancing method calls to clustered EJBs. Because clustered objects are deployed on all WebLogic Server instances in the cluster, each object instance is available locally to each server. WebLogic Server optimizes method calls to clustered EJBs by always selecting the local object instance, rather than distributing requests to remote objects and incurring additional network overhead.

This collocation strategy is, in most cases, more efficient than load balancing each method request to a different server. However, if the processing load to individual servers becomes unbalanced, it may eventually become more efficient to submit method calls to remote objects rather than process methods locally.

To utilize load balancing for method calls to clustered EJBs, you must split the presentation and object tiers of the Web application onto separate physical clusters. This is described in Recommended Multi-Tier Architecture below.

Recommended Multi-Tier Architecture

The recommended multi-tier architecture uses two separate WebLogic Server clusters: one to serve static HTTP content and clustered servlets, and one to serve clustered EJBs. The multi-tier cluster is recommended for Web applications that:

- Require load balancing for method calls to clustered EJBs.

- Require more flexibility for balancing the load between servers that provide HTTP content and servers that provide clustered objects.

- Require higher availability (fewer single points of failure).

Note: Consider the frequency of invocations from the presentation tier to the object tier when considering a multi-tier architecture. If presentation objects usually invoke the object tier, a combined tier architecture may offer better performance than a multi-tier architecture.

The following figure depicts the recommended multi-tier architecture.
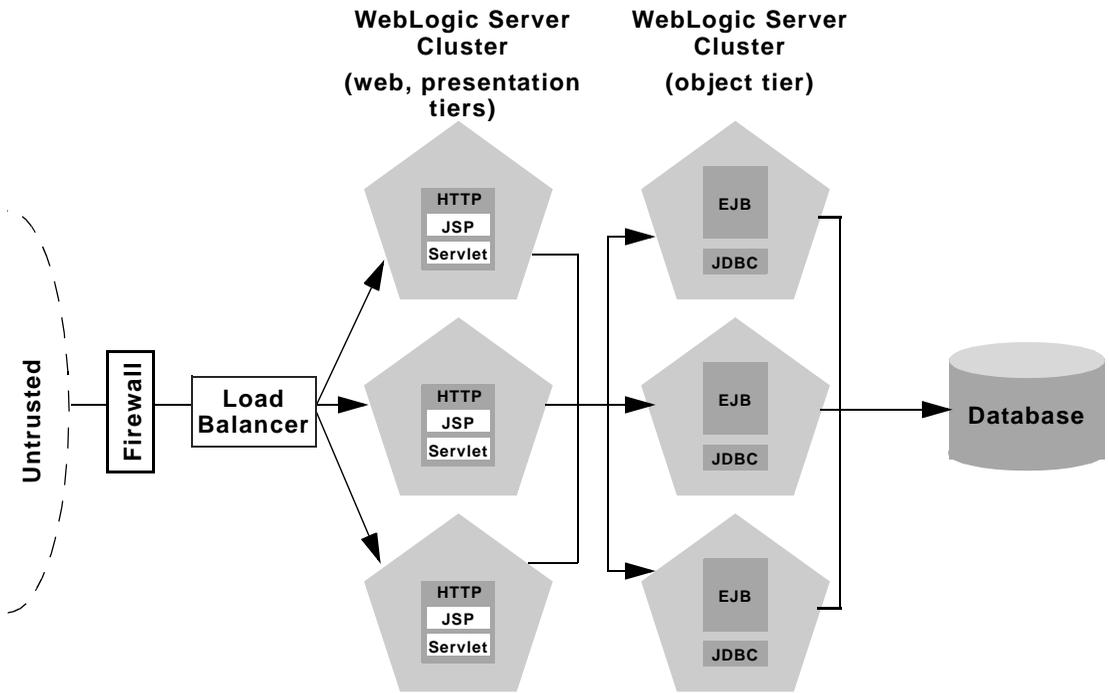
Physical Hardware and Software Layers

The advanced recommended configuration contains two physical layers of hardware and software, which comprise the logical tiers of the application itself: the web tier, presentation tier, and object tier.

Web/Presentation Layer

The web/presentation layer consists of a cluster of WebLogic Server instances dedicated to hosting static HTTP pages, servlets, and JSPs. This servlet cluster *does not* host clustered objects. Instead, servlets in the presentation tier cluster act as clients for clustered objects, which reside on an separate WebLogic Server cluster in the object layer.

Object Layer

The object layer consists of a cluster of WebLogic Server instances that hosts only clustered objects—EJBs and RMI objects as necessary for the web application. By hosting the object tier on a dedicated cluster, you lose the default collocation optimization for accessing clustered objects described in Understanding Object Clustering. However, you gain the ability to load balance on each method call to certain clustered objects, as described below.

Benefits of Multi-Tier Architecture

The multi-tier architecture provides most of the benefits of the Recommended Basic Cluster, and also introduces these strengths:

- **Load Balancing EJB Methods**: By hosting servlets and EJBs on separate clusters, servlet method calls to EJBs can be load balanced across multiple servers. This process is described in detail in Load Balancing for Clustered Object Calls below.

- **Improved Server Load Balancing**: Separating the presentation and object tiers onto separate clusters provides more options for distributing the load of the web application. For example, if the application accesses HTTP and servlet content more often than EJB content, you can use a large number of WebLogic Server instances in the presentation tier cluster to concentrate access to a smaller number of servers hosting EJBs.

- **Higher Availability**: By utilizing additional WebLogic Server instances, the multi-tier architecture has fewer points of failure than the basic cluster architecture. For example, if a WebLogic Server that hosts EJBs fails, the HTTP- and servlet-hosting capacity of the Web application is not affected.

- **Improved Security Options**: By separating the presentation and object tiers onto separate clusters, you can use a firewall policy that places only the servlet/JSP cluster in the DMZ. Servers hosting clustered objects can be further protected by denying direct access from untrusted clients. See Security Options for Cluster Architectures for more information.

Load Balancing for Clustered Object Calls

WebLogic Server's collocation optimization for clustered objects relies on having a clustered object (the EJB or RMI class) hosted on the same server instance as the replica-aware stub that calls the object.

The net effect of isolating the object tier is that no client (HTTP client, Java client, or servlet) ever acquires a replica-aware stub on the same server that hosts the clustered object. Because of this, WebLogic Server cannot use its collocation optimization, and servlet calls to clustered objects are automatically load balanced according to the logic contained in the replica-aware stub. The following figure depicts a client accessing a clustered EJB instance in the multi-tier architecture.
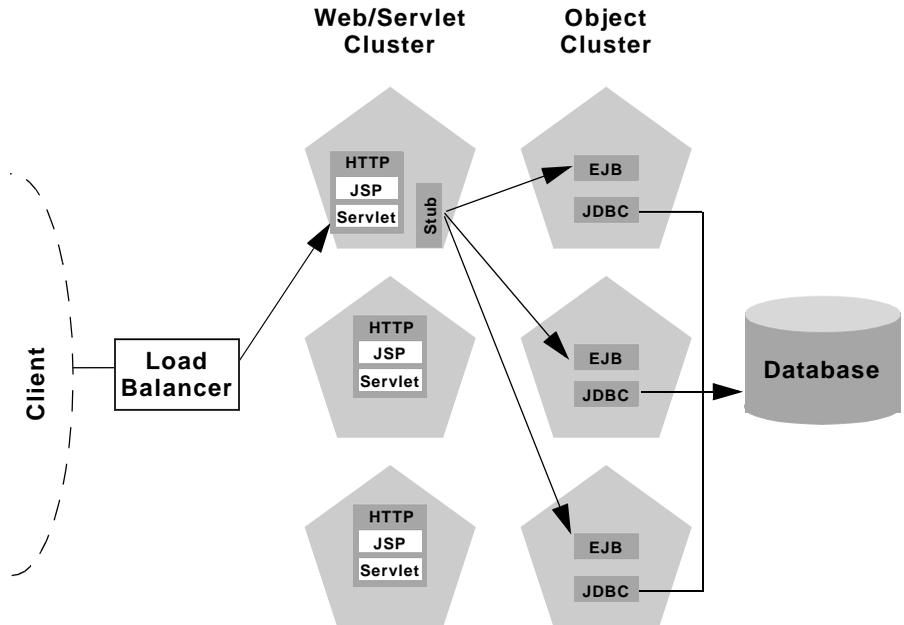
Tracing the path of the client connection, you can see the implication of isolating the object tier onto separate hardware and software:

1. An HTTP client connects to one of several WebLogic Server instances in the web/servlet cluster, going through a load balancer to reach the initial server.

2. The client accesses a servlet hosted on the WebLogic Server cluster.

3. The servlet acts as a client to clustered objects required by the web application. In the example above, the servlet accesses a stateless session EJB.

   The servlet looks up the EJB on the WebLogic Server cluster that hosts clustered objects. The servlet obtains a replica-aware stub for the bean, which lists the addresses of all servers that host the bean, as well as the load balancing logic for accessing bean replicas.

   Note: EJB replica-aware stubs and EJB home load algorithms are specified using elements of the EJB deployment descriptor. See weblogic-ejb-jar.xml Deployment Descriptors for more information.

4. When the servlet next accesses the EJB (for example, in response to another client), it uses the load-balancing logic present in the bean's stub to locate a replica. In the example above, multiple method calls are directed using the round-robin algorithm for load balancing.

In this example, if the same WebLogic Server cluster hosted both servlets and EJBs (as in the Recommended Basic Cluster), WebLogic Server would not load balance requests for the EJB. Instead, the servlet would always invoke methods on the EJB replica hosted on the local server. Using the local EJB instance is more efficient than making remote method calls to an EJB on another server. However, the multi-tier architecture enables remote EJB access for applications that require load balancing for EJB method calls.

Configuration Notes for Multi-Tier Architecture

Because the multi-tier architecture provides load balancing for clustered object calls, the system generally utilizes more IP sockets than a combined-tier architecture. In particular, during peak socket usage, each WebLogic Server in the cluster that hosts servlets and JSPs may potentially use a maximum of:

■ One socket for replicating HTTP session states between primary and secondary servers, plus

■ One socket for each WebLogic Server in the EJB cluster, for accessing remote objects

For example, in the figure shown in Recommended Multi-Tier Architecture, each server in the servlet/JSP cluster could potentially open a maximum of five sockets. This maximum represents a worst-case scenario where primary and secondary session states are equally dispersed throughout the servlet cluster, and each server in the servlet cluster simultaneously accesses a remote object on each server in the object cluster. In most cases, the number of sockets actual sockets in use would be less than this maximum.

If you use a pure-Java sockets implementation with the multi-tier architecture, ensure that you configure enough socket reader threads to accommodate the maximum potential socket usage. See Configuring Reader Threads for Java Socket Implementation for more details.

Because the multi-tier architecture uses a hardware load balancer, you must configure the load balancer to maintain a "sticky" connection to the client's point-of-contact server if you use in-memory session state replication. See Configure Load Balancing Hardware (Optional) for more information.

Limitations of Multi-Tier Architecture

Because the advanced configuration cannot optimize object calls using the collocation strategy, the Web application incurs network overhead for all method calls to clustered objects. This overhead may be acceptable, however, if your Web application requires any of the benefits described in Benefits of Multi-Tier Architecture.

For example, if your Web clients make heavy use of servlets and JSPs but access a relatively small set of clustered objects, the multi-tier architecture enables you to concentrate the load of servlets and object appropriately. You may configure a servlet cluster of ten WebLogic Server instances and an object cluster of three WebLogic Server instances, while still fully utilizing each server's processing power.

Firewall Restrictions

If you place a firewall between the servlet cluster and object cluster in a multi-tier architecture, you must bind all servers in the object cluster to public DNS names, rather than IP addresses. Binding those servers with IP addresses can cause address translation problems and prevent the servlet cluster from accessing individual server instances.

If the internal and external DNS names of a WebLogic Server instance are not identical, use the `ExternalDNSName` attribute for the server instance to define the server's external DNS name. Outside the firewall the `ExternalDNSName` should translate to external IP address of the server. Set this attribute in the Administration Console using the Server—>Configuration—>General tab. See Server—>Configuration—>General in Administration Console Online Help.

Recommended Proxy Architectures

You can also configure WebLogic Server clusters to operate alongside existing Web servers. In this architecture, a bank of Web servers provides static HTTP content for the Web application, using a WebLogic proxy plug-in or `HttpClusterServlet` to direct servlet and JSP requests to a cluster.

Two-Tier Proxy Architecture

The two-tier proxy architecture is similar to the Recommended Basic Cluster, except that static HTTP servers are hosted on a bank of Web servers.
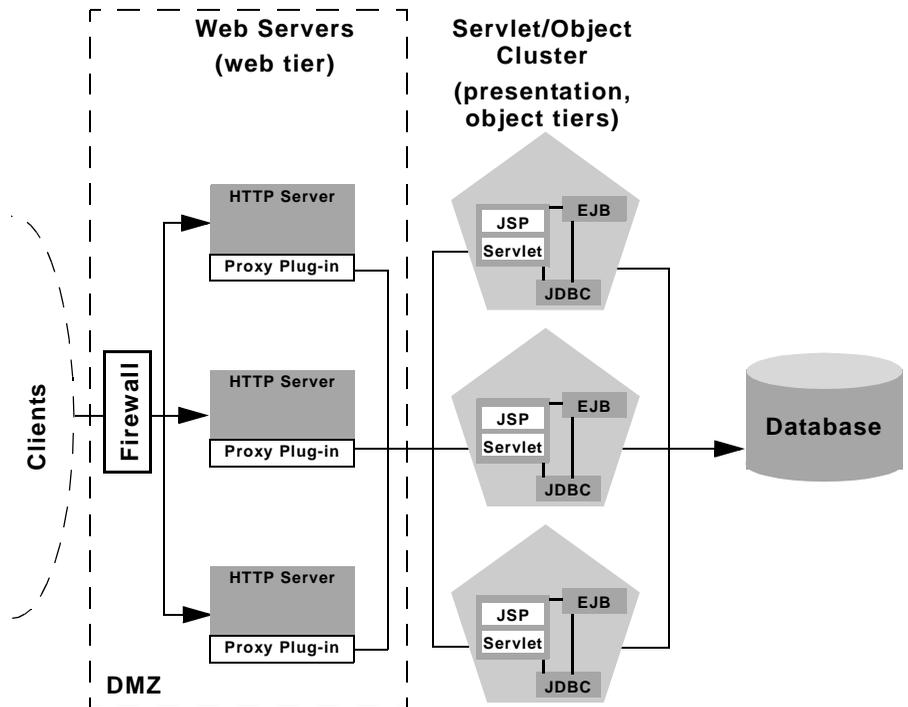
Physical Hardware and Software Layers

The two-tier proxy architecture contains two physical layers of hardware and software.

Web Layer

The proxy architecture utilizes a layer of hardware and software dedicated to the task of providing the application's web tier. This physical web layer can consist of one or more identically-configured machines that host one of the following application combinations:

- WebLogic Server with the HttpClusterServlet

- Apache with the WebLogic Server Apache proxy plug-in

- Netscape Enterprise Server with the WebLogic Server NSAPI proxy plug-in

■ Microsoft Internet Information Server with the WebLogic Server Microsoft-IIS proxy plug-in

Regardless of which Web server software you select, keep in mind that the physical tier of Web servers should provide only static Web pages. Dynamic content—servlets and JSPs—are proxied via the proxy plug-in or `HttpClusterServlet` to a WebLogic Server cluster that hosts servlets and JSPs for the presentation tier.

Servlet/Object Layer

The recommended two-tier proxy architecture hosts the presentation and object tiers on a cluster of WebLogic Server instances. This cluster can be deployed either on a multihomed machine or on multiple separate machines.

The Servlet/Object layer differs from the combined-tier cluster described in Recommended Basic Cluster in that it does not provide static HTTP content to application clients.

Multi-Tier Proxy Architecture

You can also use a bank of Web servers as the front-end to a pair of WebLogic Server clusters that host the presentation and object tiers. This architecture is shown in the figure below.
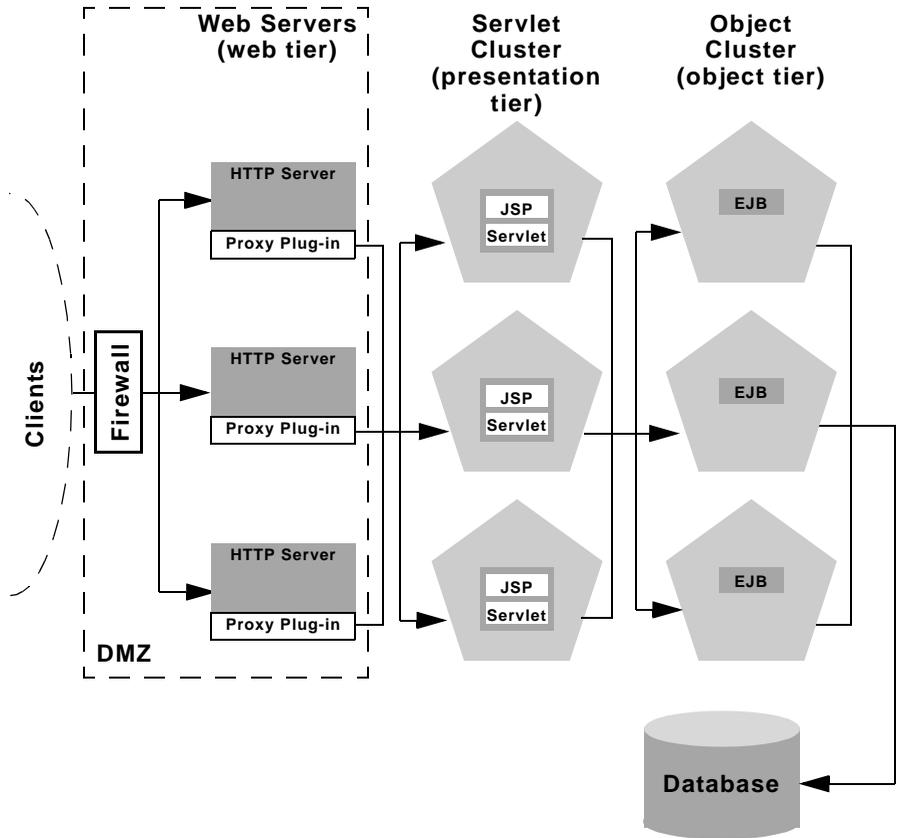
This architecture provides the same benefits (and the same limitations) as the Recommended Multi-Tier Architecture. It differs only insofar as the web tier is placed on a separate bank of Web servers that utilize WebLogic proxy plug-ins.

Proxy Architecture Trade-Offs

Using standalone Web servers and proxy plug-ins provides the following advantages:

- **Utilize Existing Hardware**: If you already have a Web application architecture that provides static HTTP content to clients, you can easily integrate existing Web servers with one or more WebLogic Server clusters to provide dynamic HTTP and clustered objects.

- **Familiar Firewall Policies:** Using a Web server proxy at the front-end of your Web application enables you to use familiar firewall policies to define your DMZ. In general, you can continue placing the Web servers in your DMZ while disallowing direct connections to the remaining WebLogic Server clusters in the architecture. The figures above depict this DMZ policy.

Using standalone Web servers and proxy plug-ins limits your Web application in the following ways:

- **Additional administration**: The Web servers in the proxy architecture must be configured using third-party utilities, and do not appear within the WebLogic Server administrative domain. You must also install and configure WebLogic proxy plug-ins to the Web servers in order to benefit from clustered servlet access and failover.

- **Limited Load Balancing Options**: When you use proxy plug-ins or the `HttpClusterServlet` to access clustered servlets, the load balancing algorithm is limited to a simple round-robin strategy.

Proxy Plug-In Versus Load Balancer

Using a load balancer directly with a WebLogic Server cluster provides several benefits over proxying servlet requests. First, using WebLogic Server with a load balancer requires no additional administration for client setup—you do not need to set up and maintain a separate layer of HTTP servers, and you do not need to install and configure one or more proxy plug-ins. Removing the Web proxy layer also reduces the number of network connections required to access the cluster.

Using load balancing hardware provides more flexibility for defining load balancing algorithms that suit the capabilities of your system. You can use any load balancing strategy (for example, load-based policies) that your load balancing hardware supports. With proxy plug-ins or the `HttpClusterServlet`, you are limited to a simple round-robin algorithm for clustered servlet requests.

Note, however, that using a third-party load balancer may require additional configuration if you use in-memory session state replication. In this case, you must ensure that the load balancer maintains a "sticky" connection between the client and its point-of-contact server, so that the client accesses the primary session state information. When using proxy plug-ins, no special configuration is necessary because the proxy automatically maintains a sticky connection.

Administration Server Considerations

To start up WebLogic Server instances that participate in a cluster, each server must be able to connect to the administration server that stores configuration information for the cluster itself. For security purposes, the administration server should reside within the same DMZ as the WebLogic Server cluster.

The administration server maintains the configuration information for all server instances that participate in the cluster. The config.xml file that resides on the administration server provides a single repository for all clustered instances (and other managed instances) in the administration server's domain. You *do not* create a separate configuration file for each server in the cluster, as with previous WebLogic Server versions.

The administration server must be available in order for clustered WebLogic Server instances to start up. Note, however, that once a cluster is running, a failure of the administration server does not affect ongoing cluster operation.

The administration server should be dedicated to the process of administering servers: maintaining configuration data, starting and shutting down servers, and deploying and undeploying applications. If the administration server also handles client requests, there is a risk of delays in accomplishing administration tasks.

There is no benefit in clustering an administration server; the administrative objects are not clusterable, and will not failover to another cluster member if the administrative server fails. Deploying applications on an administration server can reduce the stability of the server and the administrative functions it provides. If an application you deploy on the administration server behaves unexpectedly, it could interrupt operation of the admininistration server.

Ensure that the administration server's IP address is not included in the cluster-wide DNS name.

What Happens if the Administration Server Fails?

The failure of an Administration Server for a domain does not affect the operation of Managed Servers in the domain. If an Administration Server for a domain becomes unavailable while the server instances it manages—clustered or otherwise—are up and running, those Managed Servers continue to run. If the domain contains clustered server instances, the load balancing and failover capabilities supported by the domain configuration remain available, even if the Administration Server fails.

Note: If an Administration Server fails because of a hardware or software failure on its host machine, other server instances on the same machine may be similarly affected. However, the failure of an Administration Server itself does not interrupt the operation of Managed Servers in the domain.

For instructions on re-starting an Administration Server, see "Restarting an Administration Server When Managed Servers are Running" in *WebLogic Server Administration Guide.*

Security Options for Cluster Architectures

The boundaries between physical hardware/software layers in the recommended configurations provide potential points for defining your Web application's De-Militarized Zone (DMZ). However, not all boundaries can support a physical firewall, and certain boundaries can support only a subset of typical firewall policies.

The sections that follow describe several common ways of defining your DMZ to create varying levels of application security.

Basic Firewall for Proxy Architectures

The basic firewall configuration uses a single firewall between untrusted clients and the Web server layer, and it can be used with either the combined-tier or multi-tier cluster architectures.
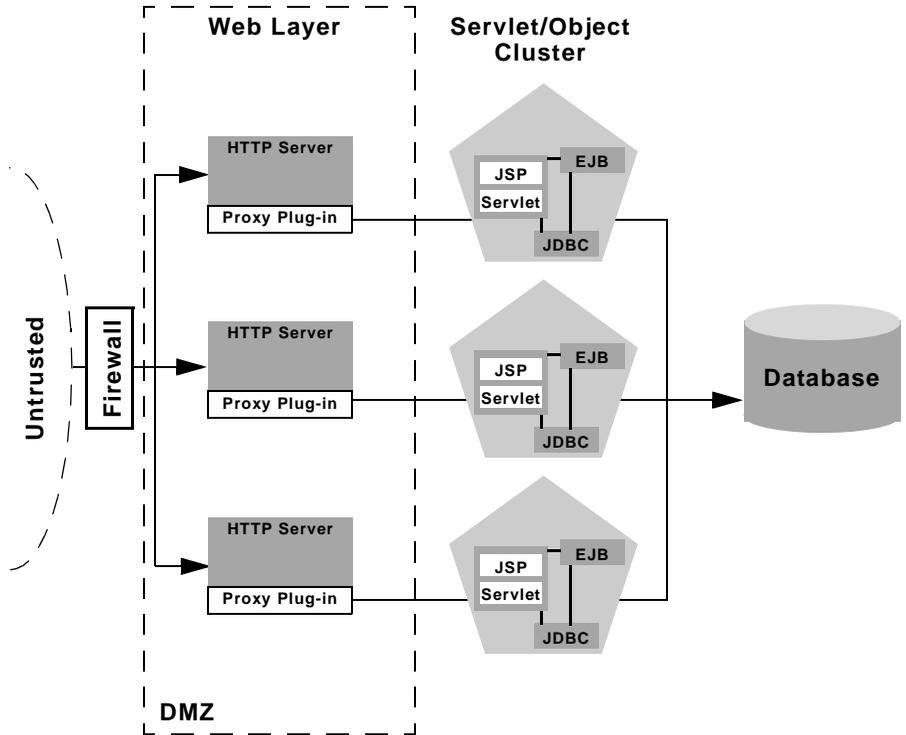
In the above configuration, the single firewall can use any combination of policies (application-level restrictions, NAT, IP masquerading) to filter access to three HTTP servers. The most important role for the firewall is to deny direct access to any other servers in the system. In other words, the servlet layer, the object layer, and the database itself must not be accessible from untrusted clients.

Note that you can place the physical firewall either in front of or behind the Web servers in the DMZ. Placing the firewall in front of the Web servers simplifies your firewall policies, because you need only permit access to the web servers and deny access to all other systems.

Note: If you place the firewall between the three Web servers and the WebLogic Server cluster, you must bind all server instances using publicly-listed DNS names, rather than IP addresses. Doing so ensures that the proxy plug-ins can freely connect to each server in the cluster and not encounter address translation errors as described in Firewall Considerations for Clusters.

If the internal and external DNS names of a WebLogic Server instance are not identical, use the ExternalDNSName attribute for the server instance to define the server's external DNS name. Outside the firewall the externalDNSName should translate to external IP address of the server. Set this attribute in the Administration Console using the Server—>Configuration—>General tab. See Server—>Configuration—>General in Administration Console Online Help.

DMZ with Basic Firewall Configurations

By denying access to all but the Web server layer, the basic firewall configuration creates a small-footprint DMZ that includes only three Web servers. However, a more conservative DMZ definition might take into account the possibility that a malicious client may gain access to servers hosting the presentation and object tiers.

For example, assume that a hacker gains access to one of the machines hosting a Web server. Depending on the level of access, the hacker may then be able to gain information about the proxied servers that the Web server accesses for dynamic content.

If you choose to define your DMZ more conservatively, you can place additional firewalls using the information in Additional Security for Shared Databases.

Combining Firewall with Load Balancer

If you use load balancing hardware with a recommended cluster configuration, you must decide how to deploy the hardware in relationship to the basic firewall. Although many hardware solutions provide security features in addition to load balancing services, most sites rely on a firewall as the first line of defense for their Web applications. In general, firewalls provide the most well-tested and familiar security solution for restricting web traffic, and should be used in front of load balancing hardware, as shown below.
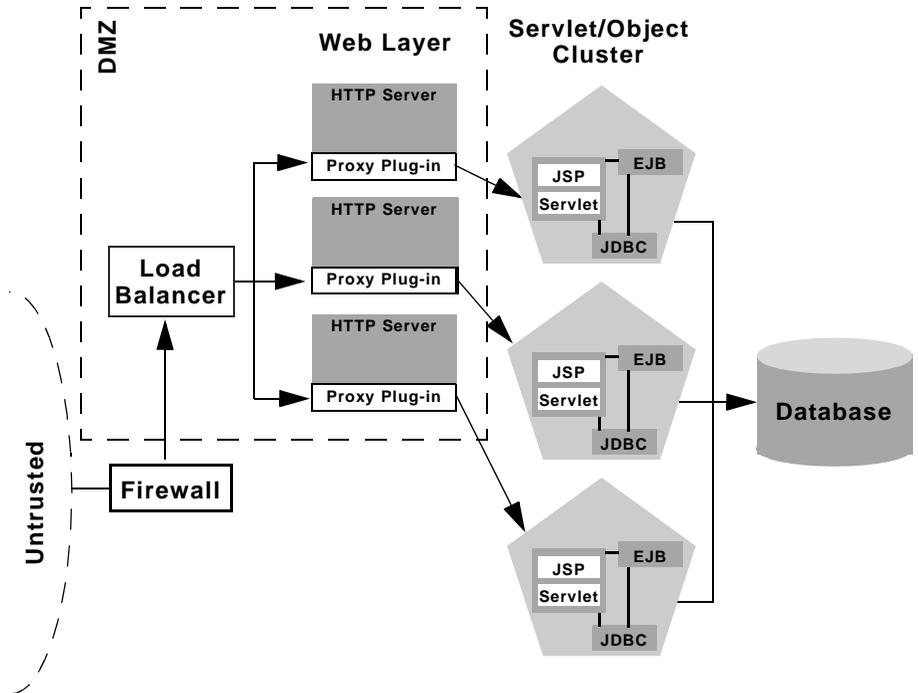
The above setup places the load balancer within the DMZ along with the web tier. Using a firewall in this configuration can simplify security policy administration, because the firewall need only limit access to the load balancer. This setup can also simplify administration for sites that support internal clients to the Web application, as described below.

Expanding the Firewall for Internal Clients

If you support internal clients that require direct access your Web application (for example, remote machines that run proprietary Java applications), you can expand the basic firewall configuration to allow restricted access to the presentation tier. The way in which you expand access to the application depends on whether you treat the remote clients as trusted or untrusted connections.

If you use a Virtual Private Network (VPN) to support remote clients, the clients may be treated as trusted connections and can connect directly to the presentation tier going through a firewall. This configuration is shown below.
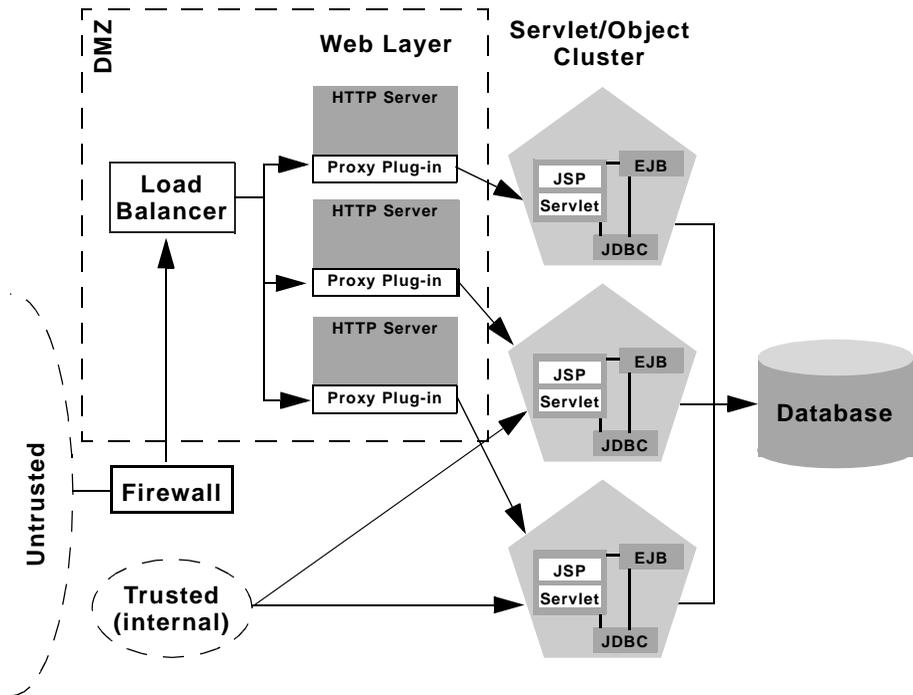
If you do not use a VPN, all connections to the Web application (even those from remote sites using proprietary client applications) should be treated as untrusted connections. In this case, you can modify the firewall policy to permit application-level connections to WebLogic Servers hosting the presentation tier, as shown below.
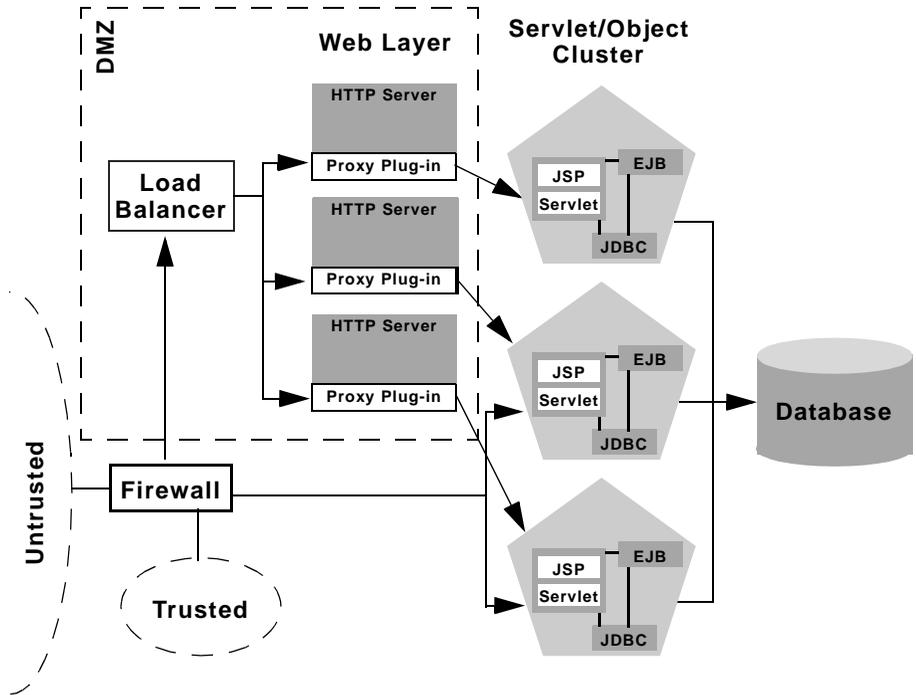
Additional Security for Shared Databases

If you use a single database that supports both internal data and data for
externally-available Web applications, you should consider placing a hard boundary
between the object layer that accesses your database. Doing so simply reinforces the
DMZ boundaries described in Basic Firewall for Proxy Architectures by adding an
additional firewall.

DMZ with Two Firewall Configuration

The following configuration places an additional firewall in front of a database server
that is shared by the Web application and internal (trusted) clients. This configuration
provides additional security in the unlikely event that the first firewall is breached, and

a hacker ultimately gains access to servers hosting the object tier. Note that this circumstance should be extremely unlikely in a production environment—your site should have the capability to detect and stop a malicious break-in long before a hacker gains access to machines in the object layer.
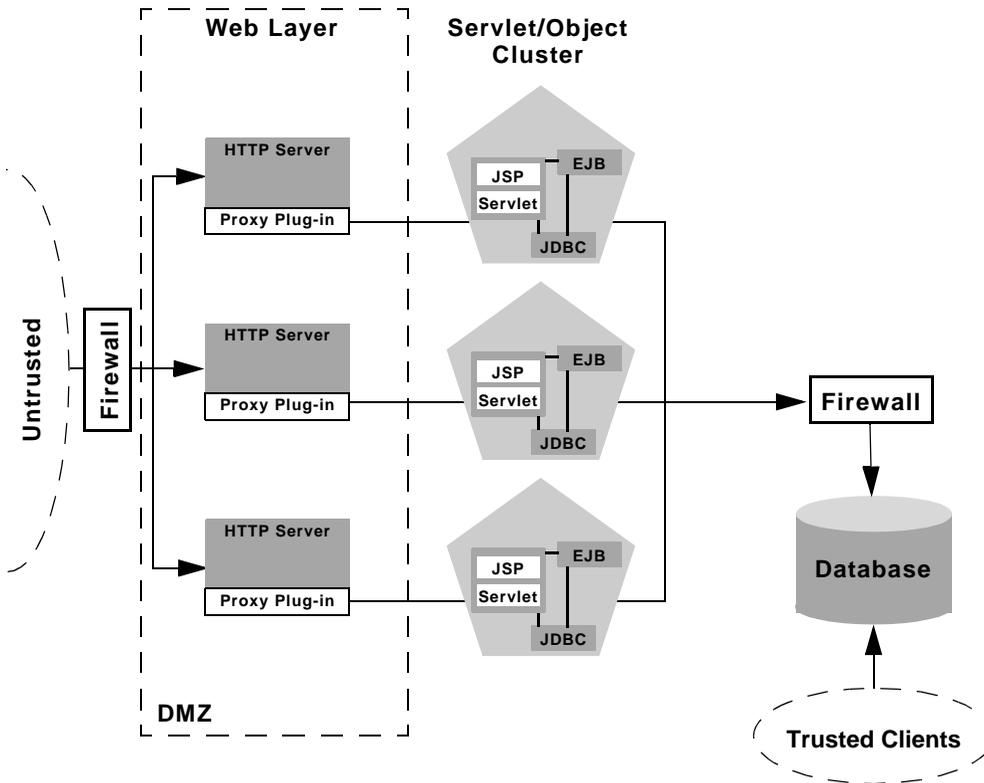
In the above configuration, the boundary between the object tier and the database is hardened using an additional firewall. The firewall maintains a strict application-level policy that denies access to all connections except JDBC connections from WebLogic Servers hosting the object tier.

Firewall Considerations for Clusters

In any cluster architecture that utilizes one or more firewalls, it is critical to identify all WebLogic Server instances using publicly-available DNS names, rather than IP addresses. Using DNS names avoid problems associated with address translation policies used to mask internal IP addresses from untrusted clients.

The following figure describes the potential problem with using IP addresses to identify WebLogic Server instances. In this figure, the firewall translates external IP requests for the subnet "xxx" to internal IP addresses having the subnet "yyy."
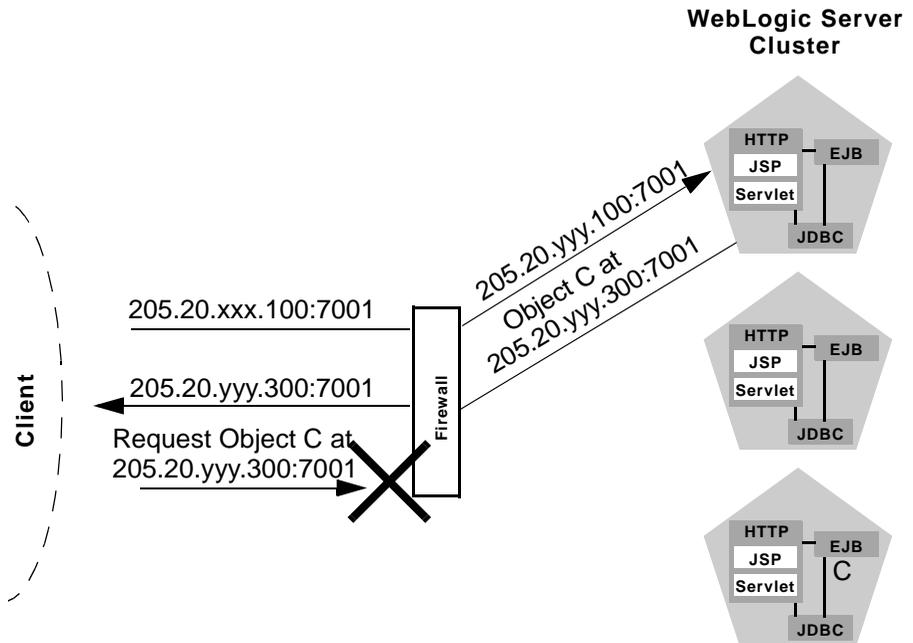
The following steps describe the connection process and potential point of failure:

1. The client initiates contact with the WebLogic Server cluster by requesting a connection to the first server at 205.20.xxx.100:7001. The firewall translates this address and connects the client to the internal IP address of 205.20.yyy.100:7001.

2. The client performs a JNDI lookup of a pinned Object C that resides on the third WebLogic Server instance in the cluster. The stub for Object C contains the *internal* IP address of the server hosting the object, 205.20.yyy.300:7001.

3. When the client attempts to instantiate Object C, it requests a connection to the server hosting the object using IP address 205.20.yyy.300:7001. The firewall denies this connection, because the client has requested a restricted, internal IP address, rather than the publicly-available address of the server.

If there was no translation between external and internal IP addresses, the firewall would pose no problems to the client in the above scenario. However, most security policies involve hiding (and denying access to) internal IP addresses.

To avoid problems in all cases, bind WebLogic Server instances to DNS names and use the same DNS names both inside and outside of the firewall. If the internal and external DNS names of a WebLogic Server instance are not identical, use the `ExternalDNSName` attribute for the server instance to define the server's external DNS name. Outside the firewall the `ExternalDNSName` should translate to external IP address of the server. Set this attribute in the Administration Console using the Server—>Configuration—>General tab. See Server—>Configuration—>General in Administration Console Online Help.

6 Administering WebLogic Clusters

The following sections contain guidelines and instructions for configuring a WebLogic Server cluster:

- "Before You Start" on page 6-1

- "Cluster Implementation Procedures" on page 6-7

Before You Start

This section summarizes prerequisite tasks and information for setting up a WebLogic Server cluster.

Obtain a Cluster Licence

Installations for clustered WebLogic Server instances must have a valid cluster license. If you do not have a cluster license, contact your BEA sales representative.

Understand the Configuration Process

The information in this section will be most useful to you if you have a basic understanding of the cluster configuration process and how configuration tasks are accomplished.

You perform all configuration for a WebLogic Server 6.1 cluster using the Administration Console. The configuration information for the cluster in stored in the config.xml file for the domain that contains the cluster. The Administration Console also manages deployment of objects and Web applications to members of the cluster.

For general information about configuring WebLogic Server 6.1, see *WebLogic Server Administration Guide*.

Determine Your Cluster Architecture

Determine what cluster architecture best suits your needs. Key architectural decisions include:

- Should you combine all application tiers in a single cluster or segment your application tiers in separate clusters?

- How will you balance the load among server instances in your cluster? Will you:
  - Use basic WebLogic Server load balancing,
  - Implement a third-party load balancer, or
  - Deploy the Web tier of your application on one or more secondary HTTP servers, and proxy requests to it?

- Should you define your Web applications De-Militarized Zone (DMZ) with one or more firewalls?

To guide these decisions, see "Planning WebLogic Server Clusters" on page 5-1.

The architecture you choose affects how you set up your cluster. The cluster architecture may also require that you install or configure other resources, such as load balancers, HTTP servers, and proxy plug-ins.

Consider Your Network and Security Topologies

Your security requirements form the basis for designing the appropriate security topology. For a discussion of several alternative architectures that provide varying levels of application security, see "Security Options for Cluster Architectures" on page 5-19.

Notes: Some network topologies can interfere with multicast communication. If you are deploying a cluster across a WAN, see "Multicast Requirements for WAN Clustering" on page 2-3.

Avoid deploying server instances in a cluster across a firewall. For a discussion of the impact of tunneling multicast traffic through a firewall, see "Firewalls Can Break Multicast Communication" on page 2-4.

Choose Machines for the Cluster Installation

Identify the machine or machines where you plan to install WebLogic Server—throughout this section we refer to such machines as "hosts"—and ensure that they have the resources required. System and software prerequisites are listed in "Preparing to Install WebLogic Server" in *Installing BEA WebLogic Server*.

Note: Do not install WebLogic Server on machines that have dynamically assigned IP addresses.

WebLogic Server Instances on Multi-CPU machines

BEA WebLogic Server has no built-in limit for the number of server instances that can reside in a cluster. Large, multi-processor servers such as Sun Microsystems, Inc. Sun Enterprise 10000 can host very large clusters or multiple clusters.

In most cases, WebLogic Server clusters scale best when deployed with one WebLogic Server instance for every two CPUs. However, you should test the actual deployment with your target Web applications to determine the optimal number and distribution of server instances. For additional information, see "Performance Considerations for Multi-CPU Machines" in *BEA WebLogic Server Performance and Tuning*.

Check Host Machines' Socket Reader Implementation

For best socket performance, configure WebLogic Server host machines to use the native socket reader implementation for your operating system, rather than the pure-Java implementation. To understand why, and for instructions for configuring native sockets or optimizing pure-Java socket communications, see "Peer-to-Peer Communication Using IP Sockets" on page 2-5.

Identify Names and Addresses

During the cluster configuration process, you supply addressing information—IP addresses or DNS names—for the cluster and its members.

For information on intra-cluster communication, and how it enables load balancing and failover, see "Server Communication in a Cluster" on page 2-2.

When you set up your cluster, you must provide location information for:

■ Administration Server

■ Managed Servers

■ Multicast location

Read the sections that follow for an explanation of the information you must provide, and factors that influence the method you use to identify resources.

Avoiding Listen Address Problems

As you configure a cluster, you can specify address information for the cluster, and the server instances that comprise it, using either IP addresses or DNS names.

DNS Names or IP Addresses?

Consider the purpose of the cluster when deciding whether to use DNS names or IP addresses. For production environments, the use of DNS names is generally recommended. The use of IP addresses can result in translation errors if:

■ Clients will connect to the cluster through a firewall, or

■ You have a firewall between the presentation and object tiers, for example, you
   have a servlet cluster and EJB cluster with a firewall in between, as described in
   the recommended multi-tier cluster.

You can avoid translation errors by binding the address of an individual server instance
to a DNS name. Make sure that a server instance's DNS name is identical on each side
of firewalls in your environment, and do not use a DNS name that is also the name of
an NT system on your network.

For more information about using DNS names instead of IP addresses, see "Firewall
Restrictions" on page 5-12.

When Internal and External DNS Names Vary

If the internal and external DNS names of a WebLogic Server instance are not
identical, use the `ExternalDNSName` attribute for the server instance to define the
server's external DNS name. Outside the firewall the `ExternalDNSName` should
translate to external IP address of the server instance. Set this attribute in the
Administration Console using the Server—>Configuration—>General tab. See
Server—>Configuration—>General in Administration Console Online Help.

Localhost Considerations

If you identify a server instance's Listen Address as localhost, non-local processes will
not be able to connect to the server instance. Only processes on the machine that hosts
the server instance will be able to connect to the server instance. If the server instance
must be accessible as localhost (for instance, if you have administrative scripts that
connect to localhost), and must also be accessible by remote processes leave the Listen
Address blank. The server instance will determine the address of the machine and
listen on it.

Assigning Names to WebLogic Server Resources

Make sure that each configurable resource in your WebLogic Server environment has
a unique name. Each domain, server, machine, cluster, JDBC connection pool, virtual
host, or other resource must have a unique name.

Administration Server Address and Port

Identify the DNS name or IP address and Listen Port of the Administration Server you
will use for the cluster.

The Administration Server is the WebLogic Server instance used to configure and manage all the WebLogic Server instances in its domain. When you start a server instance that belongs to a cluster, you identify the host and port of its Administration Server.

Managed Server Addresses and Listen Ports

Identify the DNS name or IP address of each Managed Server planned for your cluster.

Each Managed Server in a cluster must have a unique IP address and the same Listen Port number. A machine that hosts multiple clustered server instances must be multihomed to provide an IP address for each server instance.

Cluster Multicast Address and Port

Identify the address and port you will dedicate to multicast communications for your cluster.

Server instances in a cluster communicate with each other using multicast—they use multicast to announce their services, and to issue periodic heartbeats that indicate continued availability.

The multicast address for a cluster should not be used for any purpose other than cluster communications. If the machine where the cluster multicast address exists hosts or is accessed by cluster-external programs that use multicast communication, make sure that those multicast communications use a different port than the cluster multicast port.

Multicast and Multiple Clusters

Multiple clusters on a network may share a multicast address and multicast port combination if necessary.

Multicast and Multi-Tier Clusters

If you are setting up the Recommended Multi-Tier Architecture, described in Chapter 5, "Planning WebLogic Server Clusters," with a firewall between the clusters, you will need two dedicated multicast addresses: one for the presentation (servlet) cluster and one for the object cluster. Using two multicast addresses ensures that the firewall does not interfere with cluster communication.

Cluster Address

When you configure a cluster using the Administration Console, you supply a cluster address that identifies the Managed Servers in the cluster.

The cluster address is used in entity and stateless beans to construct the host name portion of URLs. If the cluster address is not set, EJB handles may not work properly.

The cluster address can be specified as:

- A DNS name that maps to the DNS names or IP addresses of the Managed Servers in the cluster.

- A list that contains the DNS name or IP address of each Managed Server in the cluster, as shown in the examples below:

```
DNSName1,DNSName2,DNSName3
```

```
IPaddress1,IPaddress2;IPaddress3
```

For production environments, BEA recommends that you set the cluster address to a DNS name that contains the host names or addresses of the Managed Servers in the cluster. If you do not define the cluster address as a DNS name that maps to the Managed Servers in the cluster, failover will not work for entity beans and EJB handles, as described in "Failover for Entity Beans and EJB Handles" on page 4-4.

Defining the cluster address as a comma-separated list of addresses is not recommended for production environments.

Note: The Administration Server should not participate in a cluster. Ensure that the Administration Server's IP address *is not* included in the cluster-wide DNS name. For more information, see "Administration Server Considerations" on page 5-18.

Cluster Implementation Procedures

This section describes how to get a clustered application up and running, from installation of WebLogic Server through initial deployment of application components.

Configuration Roadmap

We present a typical sequence of tasks for a cluster implementation, and highlight key considerations that influence your configuration choices. The exact process you follow is driven by the unique characteristics of your environment and the nature of your application. These are the tasks described in this section:

- "Install WebLogic Server" on page 6-8

- "Define Machine Names (Optional)" on page 6-9

- "Create WebLogic Server Instances" on page 6-10

- "Create a New Cluster" on page 6-11

- "Configure Load Balancing Hardware (Optional)" on page 6-13

- "Configure Proxy Plug-Ins (Optional)" on page 6-16

- "Configure Replication Groups (Optional)" on page 6-17

- "Configure Clustered JDBC" on page 6-17

- "Configure JMS" on page 6-19

- "Deploy Web Applications and EJBs" on page 6-21

- "Configure In-Memory HTTP Replication" on page 6-20

- "Additional Configuration Topics" on page 6-21

Not every step is required for every cluster implementation. Additional steps may be necessary in some cases.

Install WebLogic Server

If you have not already done so, install the WebLogic Server product. For multihomed cluster installations, you can install a single WebLogic Server distribution under the `/bea` directory to use for all clustered instances. For remote, networked machines, perform the installation on each WebLogic Server host.

Installations for clustered WebLogic Server instances must also have a valid cluster license. See "Obtain a Cluster Licence" on page 6-1 for more details.

Note: Do not use a shared filesystem and a single installation to run multiple WebLogic Server instances on separate machines. Using a shared filesystem introduces a single point of contention for the cluster. All servers must compete to access the filesystem (and possibly to write individual log files). Moreover, should the shared filesystem fail, you may be unable to start clustered servers.

Define Machine Names (Optional)

WebLogic Server uses configured machine names to determine whether or not two server instances reside on the same physical hardware. Machine names are generally used with multihomed machines that host WebLogic Server instances. If you do not define machine names for such installations, each instance is treated as if it resides on separate physical hardware. This can negatively affect the selection of servers to host secondary HTTP session state replicas, as described in "Using Replication Groups" on page 3-5.

Before creating new WebLogic Server instances, use the following instructions to define the names of individual machines that will host the server instances:

1. Boot the Administration Server for your system. For instructions, see "Starting the WebLogic Administration Server" in *WebLogic Server Administration Guide*.

2. Start the administration console using the instructions in "Starting the Administration Console" in *WebLogic Server Administration Guide*.

3. Select the Machines node.

4. Select Create a new Machine... to define a Windows NT machine, or select Create a new UNIX Machine...

5. Enter a unique name for the new machine in the Name attribute field.

6. Click Create to create the new machine definition.

7. If you wish to configure other attributes for a new UNIX server, refer to the Administration Console Online Help.

8. Repeat these steps for each machine that will host one or more WebLogic Server instances in the cluster.

Create WebLogic Server Instances

Before servers can join a cluster, you must create new definitions for each server instance using the WebLogic Server Administration Console. Follow these steps:

1. Boot the Administration Server for your system. For instructions, see "Starting the WebLogic Administration Server" in *WebLogic Server Administration Guide*.

2. Start the administration console using the instructions in "Starting the Administration Console" in *WebLogic Server Administration Guide*.

3. Select the Servers node.

4. Select Configure a new Server.

5. Enter values for the following attribute fields:

   • Name: Enter a name to use for this server instance in the Administration Console. You will use this name in the startup command to indicate which server you are booting.

     See "Assigning Names to WebLogic Server Resources" on page 6-5.

   • Listen Port: Enter the port number used to connect to this server. All servers in a given cluster *must* use the same port number.

   • Listen Address: Enter the DNS name or IP address to bind to this server.

     See "Avoiding Listen Address Problems" on page 6-4.

6. For the Machine attribute, select the machine on which the new server resides. The Machine attribute lists all machine names that you created in "Define Machine Names (Optional)" on page 6-9".

7. Click Create to create the new server configuration.

8. If you wish to configure other attributes of the new server, refer to "Server Configuration Tasks" in *WebLogic Server Administration Guide*.

9. Repeat these steps for each WebLogic Server that will participate in the cluster.

Create a New Cluster

After you have created the individual WebLogic Server instances, follow these steps to configure a new cluster:

1. Open the Administration Console.

2. Select the Clusters node.

3. Select Configure a new Cluster.

4. Enter values for the following attribute fields:

   - Name: Enter the name to use for this cluster in the Administration Console. You will use this name to assign membership to the cluster and configure other cluster attributes.

   - Cluster Address: Enter the DNS name (containing the IP addresses or DNS names of all individual WebLogic Server instances in the cluster) to use for the cluster. For more information, see "Cluster Address" on page 6-7.

   - Default Load Algorithm: Enter the default load algorithm to use for this cluster, or accept the default.

5. Click Create to create the new cluster configuration.

6. Select the Multicast tab.

7. Enter the cluster's multicast address in the Multicast Address attribute field.

8. Apply the changes.

Starting a WebLogic Server Cluster

To start a WebLogic Server instance that participates in a cluster, you use the same procedure as you would for starting any Managed Server. You identify the Administration Server the server instance should use. The configuration information for the server instance is obtained from the `config.xml` file associated with the Administration Server.

The basic process for starting up a cluster is:

9. Start the Administration Console for the domain in which the cluster resides. For instructions, see "Starting the Administration Console" in *WebLogic Server Administration Guide*.

10. .Start individual, clustered servers instances as Managed Servers. For example:

```
% java -ms64m -mx64m -classpath $CLASSPATH
-Dweblogic.Domain=mydomain -Dweblogic.Name=clusterServer1
-Djava.security.policy==$WL_HOME/lib/weblogic.policy
-Dweblogic.management.server=192.168.0.101:7001
-Dweblogic.management.username=system
-Dweblogic.management.password=systemPassword weblogic.Server
```

The server instance's cluster configuration is stored by the Administration Server, so you do not need to explicitly include address binding or multicast information in the command line. You do, however, need to specify:

- weblogic.Name, to identify the clustered instance you want to start.

- weblogic.management.server, to identify the host and port number of the Administration Server that stores the clustered instance's configuration.

- weblogic.management.username, to specify a username to connect to the Administration Server.

- weblogic.management.password, to specify the user's password.

See "Starting a WebLogic Managed Server" in *WebLogic Server Administration Guide* for more details.

11. As the clustered instance boots, monitor the log messages to see that it joins the cluster. The instance begins by binding the multicast address and common port number of the cluster:

```
Starting Cluster Service ....

<Jul 25, 2001 6:35:17 PM PDT> <Notice> <WebLogicServer> <ListenThread listening
on port 7001, ip address 172.17.13.25>

<Jul 25, 2001 6:35:18 PM PDT> <Notice> <Cluster> <Listening for multicast messages
(cluster MyCluster) on port 7001 at address 239.0.0.84>

<Jul 25, 2001 6:35:18 PM PDT> <Notice> <WebLogicServer> <Started WebLogic Managed
Server "MyServer-1" for domain "mydomain" running in Production Mode>
```

12. To verify that all clustered instances are participating in the cluster, open the
    Administration Console. Select the cluster, click the Monitoring tab, then select
    `Monitor server participation in cluster`. If one or more of the started
    server instances do not appear in the display, verify that the server instances have
    been configured with the correct multicast address and port number.

Configure Load Balancing Hardware (Optional)

This section provides general guidelines for setting up an external load balancer. For
instructions on setting up a BIG-IP load balancer, see "Configuring BIG-IP™
Hardware with Clusters"

For information about how WebLogic Server accesses clustered servlets and JSPs in a
configuration that includes load balancing hardware, see "Accessing Clustered
Servlets and JSPs with Load Balancing Hardware" on page 3-10.

If you are using a hardware load balancing solution with HTTP session state
replication, you must configure your load balancer to support WebLogic Server
session cookies. The configuration steps depend on the type of persistence cookie
persistence mechanism used on the load balancing hardware. The following table
shows possible configurations.

Persistence Type		
Active Cookie Persistence		**Passive Cookie Persistence**
Load Balancer Overwrites Cookie	**Load Balancer Inserts Additional Cookie**	
Not supported	No configuration required	Specify offset and string constant

Using Active Cookie Persistence

WebLogic Server clusters do not support active cookie persistence mechanisms that
overwrite or modify the WebLogic HTTP session cookie.

If the load balancer's active cookie persistence mechanism works by adding its own cookie to the client session, no additional configuration is required to use the load balancer with a WebLogic Server cluster.

Using Passive Cookie Persistence

A load balancer that uses passive cookie persistence can use a string constant in the WebLogic session cookie to associate a client with the server instance hosting its primary HTTP session state. The string constant uniquely identifies a server instance in the cluster. You must configure the load balancer with the offset and length of the string constant. The correct values for the offset and length depend on the format of the session cookie.

Understanding the Session Cookie

The basic format of a session cookie is:

Random Session ID		1 byte delimiter		String Constant

where:

- *Random Session ID* is a randomly generated identifier of the HTTP session. The length of the value is configurable, by setting the IDLength parameter in the <session-descriptor> element in the weblogic.xml file for an application. By default, the length of the Random Session ID is 52 bytes.

- *String Constant* identifies the WebLogic Server instance hosting the session, including its host machine and listen ports.

  In a non-clustered environment, the string constant can be up to 60 bytes long, with this format:

  ```
  Primary_JVMID_Differentiator!HOST!PORT!SSLPORT
  ```

  In a clustered environment, the string constant specifies both the primary and the secondary WebLogic Server instances for the session—it may be up to 120 bytes long, with this format:

  ```
  Primary_JVMID_Differentiator!Host!Port!SSLPort!Secondary_JVMID_
  Differentiator!Host!Port!SSLPort
  ```

The first 19 bytes of the string constant uniquely identify the primary WebLogic Server instance for the initial release of WebLogic Server version 6.1. For WebLogic Server version 6.1 Service Pack 1 and later, the first 10 bytes are sufficient to identify the server instance.

Session Cookie for WAP-Enabled Applications

Because many wireless devices do not support cookies, an alternative method of session tracking—URL rewriting—can be used for WAP applications. Because wireless devices often support only a limited length URL, use of URL rewriting for WAP-enabled applications requires a reduction in the length of session parameters. For more information about using URL rewriting for wireless applications, see "An Alternative Session-Tracking Method—URL Rewriting" in *Programming WebLogic Server for Wireless Services.*

There are two techniques that are used to shorten session parameters that can affect how you configure your load balancer:

- The length of the Random Session ID can be shortened by editing the value of the IDLength parameter in the <session-descriptor> element in the weblogic.xml file for your application application. The minimum length for Random Session ID is 8 bytes.

- The length of the session parameters that specify the primary and secondary server instances can be shortened by setting the WAPEnabled parameter in the config.xml file for the domain to "true". When WAPEnabled is "true", the primary and secondary server instances are identified in the cookie by short hash codes.

Shortening the Random Session ID and the session parameters that specify the server instances results in a string constant with this format:

Rand_Sess_ID!Primary_JVMID_HASH!Secondary_JVMID_HASH

where the length of:

Rand_Sess_ID is 8 bytes

Primary_JVMID_HASH is 8 to 10 bytes

SECONDARY_JVMID_HASH is 8 to 10 bytes

Configuring Load Balancer

If you have not changed the length of the Random Session ID from its default length of 52 bytes, use the facilities of your load balancer to set the offset and length of the string constant as described below:

- string constant offset—set to 53 bytes, the default Random Session ID length plus 1 byte for the delimiter character.

- string constant length—BEA recommends that you set the length of the string constant to at least 19 bytes, a length sufficient to ensure that the primary server instance can be uniquely identified.

If your application or environmental requirements dictate that you change the length of the Random Session ID from its default value of 52 bytes, you must set the string constant offset on the load balancer accordingly. The offset for the string constant must equal the length of the Random Session ID plus 1 byte for the delimiter character.

If your cluster hosts a WAP-enabled application, there are session parameter considerations that might affect how you configure your load balancer. For details, see "Configuring Load Balancer for Reduced Length Session Parameters (WAP-Enabled)."

Configuring Load Balancer for Reduced Length Session Parameters (WAP-Enabled)

If `WAPEnabled` is set to "true" for your domain, and `IDLength` is 8 bytes, set the offset and length of the string constant as follows:

- String constant offset—set to 9 bytes, the default Random Session ID length plus 1 byte for the delimiter character.

- String constant length—Set the length of the string constant to 8, the minimum length of the hash code that identifies the primary server instance.

Configure Proxy Plug-Ins (Optional)

If you are using Web servers with WebLogic proxy plug-ins (or the `HttpClusterServlet`) to access the cluster, use the instructions in the *WebLogic Server Administration Guide* to configure your proxy software. Keep in mind that all

Web servers that proxy requests to the cluster must be configured identically. For information about connection and failover using the proxy plug-in, see "Accessing Clustered Servlets and JSPs Using a Proxy" on page 3-7.

Configure Replication Groups (Optional)

If your cluster will host servlets or stateful session EJBs, you may want to create replication groups of WebLogic Server instances to host the session state replicas. To do so, follow the instructions in "Using Replication Groups" on page 3-5 to determine which server instances should participate in each replication group, and to determine each server instance's preferred replication group.

To configure replication groups for a WebLogic Server instance:

1. Open the Administration Console.

2. Select the Servers node.

3. Select the server instance to configure.

4. Select the Cluster tab.

5. Enter values for the following attribute fields:

   - Replication Group: Enter the replication group name to which this server instance belongs.

   - Preferred Secondary Group: Enter the name of the replication group you would like to use to host replicated HTTP session states for this server instance.

6. Apply the changes.

Configure Clustered JDBC

This section provides instructions for configuring JDBC components using the Administration Console. The choices you make as you configure the JDBC components are reflected in the config.xml file for the WebLogic Server domain that contains the cluster.

First you create the connection pools and optionally a multipool, then you create the data source. When you create a data source object, you specify a connection pool or multipool as one of the data source attributes. This associates that data source with one specific connection pool or multipool.

- For an overview of how JDBC objects work in a WebLogic Server cluster, see "JDBC Connections" on page 1-4.

- For a description of how clustered JDBC can increase application availability, see "Failover and JDBC Connections" on page 2-21

- For a description of how clustered JDBC supports load balancing, see "Load Balancing and JDBC Connections" on page 2-18

Clustering Connection Pools

Perform these steps to set up a basic connection pool in a cluster:

1. Create a connection pool.

    For instructions, see "Configure a JDBC Connection Pool" in Administration Console Online Help.

2. Assign the connection pool to the cluster.

    For instructions, see "Assign a JDBC Connection Pool to One or More Servers or Clusters".

3. Create the data source. Specify the connection pool created in the previous step in the Pool Name attribute.

    For instructions, see "Configure a JDBC Data Source" in Administration Console Online Help

4. Assign the data source to the cluster.

    For instructions, see "Assign a JDBC Data Source".

Clustering Multipools

Perform these steps to create a clustered multipool for increased availability, and optionally, load balancing.

Note: Multipools are typically used to provide increased availability and load balancing of connections to replicated, synchronized instances of a database. For more information, see "JDBC Connections" on page 1-4.

1. Create two or more connection pools.

   For instructions, see "Configure a JDBC Connection Pool" in Administration Console Online Help.

2. Assign each connection pool to the cluster.

   For instructions, see "Assign a JDBC Connection Pool to One or More Servers or Clusters".

3. Create a multipool. Assign the connection pools created in the previous step to the multipool.

   For instructions, see "Configure a JDBC MultiPool".

4. Assign the Multipool to the cluster.

   For instructions, see "Assign a JDBC MultiPool to One or More Servers or Clusters".

5. Create the data source. Specify the multipool created in the previous step in the `Pool Name` attribute.

   For instructions, see "Configure a JDBC Data Source".

6. Assign the data source to the cluster.

   For instructions, see "Assign a JDBC Data Source".

Configure JMS

Follow the instructions in "Managing JMS" in the *Administration Guide* to configure JMS for a cluster. Adhere to the following guidelines:

■ JMS Servers—You cannot deploy a JMS server on more than one Managed Server.

■ Destinations—You cannot deploy the same destination on more than one JMS server.

- Connection Factories—Each connection factory can be deployed on multiple WebLogic Servers.

- Managed Servers—JMS clients depend on unique WebLogic Server names to successfully access clustered WebLogic Servers—even when the servers reside in different domains. Make sure that all Managed Servers that JMS clients contact have unique server names.

Configure In-Memory HTTP Replication

To support automatic failover for servlets and JSPs, WebLogic Server replicates HTTP session states in memory.

Note: WebLogic Server can also maintain the HTTP session state of a servlet or JSP using file-based or JDBC-based persistence. For more information on these persistence mechanisms, see "Configuring Session Persistence" in *Assembling and Configuring Web Applications*.

In-memory HTTP Session state replication is controlled separately for each application you deploy. The parameter that controls it—PersistentStoreType– appears within the session-descriptor element, in the WebLogic deployment descriptor file, weblogic.xml, for the application.

domain_directory/applications/*application_directory/*Web-Inf/weblogic.xml

To use in-memory HTTP session state replication across server instances in a cluster, set the PersistentStoreType to replicated. The fragment below shows the appropriate XML from weblogic.xml.

```
<session-descriptor>

      <session-param>

            <param-name> PersistentStoreType </param-name>

            <param-value> replicated </param-value>

      </session-param>

</session-descriptor>
```

Deploy Web Applications and EJBs

Use the instructions in *Packaging and Deploying Web Applications* to deploy your Web application and/or EJBs to the cluster. When you select a target for the application or EJB, choose the cluster name you specified in "Create a New Cluster" on page 6-11, rather than individual WebLogic Server instances in the cluster. Using the cluster name ensures that the application or EJB is deployed homogeneously throughout the cluster.

Clustered objects in WebLogic Server *must* be deployed homogeneously. If the object contains a replica-aware stub, use the Administration Console to deploy it using the cluster name. Otherwise, deploy non-replica-aware ("pinned") objects only to individual server instances.

The Administration Console automates deploying replica-aware objects to clusters. When you deploy an application or object to a cluster, the Administration Console automatically deploys it to all members of the cluster (whether they are local to the Administration Server machine or they reside on remote machines).

When you deploy an application with the Administration Console, the administration server sends a copy of the application file(s) to the targeted server instances, which then load the application.

Note: If deployment to a server instance fails, the deployment state across the target server instances may be inconsistent.

To learn about how clustered objects are bound in the cluster-wide JNDI tree, see "Creating the Cluster-Wide JNDI Tree" on page 2-11 and "Updating the JNDI Tree" on page 2-14.

Additional Configuration Topics

The sections below contain useful tips for particular cluster configurations.

Configure IP Sockets

For best socket performance, BEA recommends that you use the native socket reader implementation, rather than the pure-Java implementation, on machines that host WebLogic Server instances.

If you must use the pure-Java socket reader implementation for host machines, you can still improve the performance of socket communication by configuring the proper number of socket reader threads for each server instance and client machine.

■ To learn more about how IP sockets are used in a cluster, and why native socket reader threads provide best performance, see "Peer-to-Peer Communication Using IP Sockets" on page 2-5, and "Client Interaction with the Cluster-Wide JNDI Tree" on page 2-14.

■ For instructions on how to determine how many socket reader threads are necessary in your cluster, see "Determining Potential Socket Usage" on page 2-7. If you are deploying a servlet cluster in a multi-tier cluster architecture, this has an effect on how many sockets you require, as described in "Configuration Notes for Multi-Tier Architecture" on page 5-11.

The sections that follow have instructions on how to configure native socket reader threads for host machines, and how to set the number of reader threads for host and client machines.

Configure Native IP Sockets Readers on Machines that Host Server Instances

To configure a WebLogic Server instance to use the native socket reader threads implementation:

1. Open the WebLogic Server Console.

2. Select the Servers node.

3. Select the server instance to configure.

4. Select the Tuning tab.

5. Check the Enable Native IO box.

6. Apply the changes.

Set the Number of Reader Threads on Machines that Host Server Instances

By default, a WebLogic Server instance creates three socket reader threads upon booting. If you determine that your cluster system may utilize more than three sockets during peak periods, increase the number of socket reader threads:

1. Open the WebLogic Server Console.

2. Select the Servers node.

3. Select the server instance to configure.

4. Select the Tuning tab.

5. Edit the percentage of Java reader threads in the Socket Readers attribute field. The number of Java socket readers is computed as a percentage of the number of total execute threads (as shown in the Execute Threads attribute field).

6. Apply the changes.

   Note: Due to a known problem in WebLogic Server 6.1 you must also set the desired thread pool size as a command line option when starting a Managed Server. For more information and instructions, see "CR179419" in *WebLogic Server 6.1 Release Notes*.

Set the Number of Reader Threads on Client Machines

On client machines you can configure the number of socket reader threads in the Java Virtual Machine (JVM) that runs the client. Specify the socket readers by defining the `-Dweblogic.ThreadPoolSize=`*`value`* and `-Dweblogic.ThreadPoolPercentSocketReaders=`*`value`* options in the Java command line for the client.

Configure Multicast Time-To-Live (TTL)

If your cluster spans multiple subnets in a WAN, the value of the Multicast Time-To-Live (TTL) parameter for the cluster must be high enough to ensure that routers do not discard multicast packets before they reach their final destination. The Multicast TTL parameter sets the number of network hops a multicast message makes before the packet can be discarded. Configuring the Multicast TTL parameter appropriately reduces the risk of losing the multicast messages that are transmitted among server instances in the cluster.

For more information about planning your network topology to ensure that multicast messages are reliably transmitted see "Multicast Requirements for WAN Clustering" on page 2-3.

To configure the Multicast TTL for a cluster, change the Multicast TTL value in the Multicast tab for the cluster in the Administration Console. The `config.xml` excerpt below shows a cluster with a Multicast TTL value of three. This value ensures that the cluster's multicast messages can pass through three routers before being discarded:

```
<Cluster

      Name="testcluster"

      ClusterAddress="wanclust"

      MulticastAddress="wanclust-multi"

      MulticastTTL="3"

/>
```

Configure Multicast Buffer Size

If multicast storms occur because server instances in a cluster are not processing incoming messages on a timely basis, you can increase the size of multicast buffers. For information on multicast storms, see "If Multicast Storms Occur" on page 2-4.

TCP/IP kernel parameters can be configured with the UNIX ndd utility. The udp_max_buf parameter controls the size of send and receive buffers (in bytes) for a UDP socket. The appropriate value for udp_max_buf varies from deployment to deployment. If you are experiencing multicast storms, increase the value of udp_max_buf by 32K, and evaluate the effect of this change.

Do not change udp_max_buf unless necessary. Before changing udp_max_buf, read the Sun warning in the "UDP Parameters with Additional Cautions" section in the "TCP/IP Tunable Parameters" chapter in *Solaris Tunable Parameters Reference Manua*l at http://docs.sun.com/?p=/doc/806-6779/6jfmsfr7o&.

Configuration Notes for Multi-Tier Architecture

If your cluster has a multi-tier architecture, see the configuration guidelines in "Configuration Notes for Multi-Tier Architecture" on page 5-11.

Enable URL Rewriting

In its default configuration, WebLogic Server uses client-side cookies to keep track of the primary and secondary server instance that host the client's servlet session state. If client browsers have disabled cookie usage, WebLogic Server can also keep track of primary and secondary server instances using URL rewriting. With URL rewriting, both locations of the client session state are embedded into the URLs passed between the client and proxy server. To support this feature, you must ensure that URL

rewriting is enabled on the WebLogic Server cluster. For instructions on how to enable URL rewriting, see "Using URL Rewriting" in *Assembling and Configuring Web Applications*.

7 Troubleshooting Common Problems

The following section describes how to troubleshoot common problems including:

- Collecting Diagnostic Information

- Addressing Common Problems

Collecting Diagnostic Information

Before contacting BEA Technical Support for help with cluster-related problems, follow the steps in this section to collect the required diagnostic information for your system. The primary diagnostic information for cluster-related problems is a log file that contains multiple thread dumps (if applicable) from the clustered server. This log file can be helpful in diagnosing a variety of cluster-related problems, but it is especially important for addressing problems related to cluster "freezes" and deadlocks.

Note: If you experience a cluster problem that involves a deadlock between server instances or otherwise causes your cluster to "hang," *a log file that contains multiple thread dumps is a prerequisite for diagnosing your problem*.

To create the required log file, follow these steps:

1. Remove or back up any log files you may currently have. In practice you should create a new log file each time you boot a WebLogic Server instance, rather than append new sessions to a historical log file.

2. Turn on verbose Garbage Collection (GC) output for your Java VM when you start WebLogic Server. See the next step for an example command line.

3. Redirect *both* the standard error and standard output to a log file. Doing so places thread dump information in the proper context with WebLogic Server informational and error messages, and provides a more useful log for diagnostic purposes. For example:

```
% java -ms64m -mx64m -verbose:gc -classpath $CLASSPATH
-Dweblogic.domain=mydomain -Dweblogic.Name=clusterServer1
-Djava.security.policy==$WL_HOME/lib/weblogic.policy
-Dweblogic.admin.host=192.168.0.101:7001
-Dweblogic.management.username=system
-Dweblogic.management.password=systemPassword weblogic.Server >>
logfile.txt 2>&1
```

4. Continue running the WebLogic Server cluster until you have reproduced the problem.

5. For server hangs, use `kill -3` or `<Ctrl>-<Break>` to create the necessary threads dumps for diagnosing your problem. Make sure the log file contains *multiple* thread dumps on each server, with distinct intervals between thread dumps.

Note: If you are running the JRockit JVM under Linux, see "Getting a JRockit Thread Dump Under Linux" on page 7-2.

Getting a JRockit Thread Dump Under Linux

If you use the JRockit JVM under Linux, use one of the following methods to generate a thread dump.

■ Use the `weblogic.admin` `THREAD_DUMP` command. For instructions and limitations, see "THREAD_DUMP" in *WebLogic Server Administration Guide.*

■ If the JVM's management server is enabled (by starting the JVM with the `-Xmanagement` option), you can generate a thread dump using the JRockit Management Console.

■ Use `Kill -3` *PID*, where *PID* is the root of the process tree.

  To obtain the root PID, perform a:

  `ps -efHl | grep 'java' **. **`

using a `grep` argument that is a string that will be found in the process stack that matches the server startup command. The first PID reported will be the root process, assuming that the `ps` command has not been piped to another routine.

Note: Under Linux, each execute thread appears as a separate process under the Linux process stack. To use Kill -3 on Linux you supply must match PID of the main WebLogic execute thread, otherwise no thread dump will be produced.

Providing Diagnostics to BEA Technical Support

After you have created a diagnostic log file (with thread dumps, if applicable), use the following guidelines when providing information to your BEA Technical Support representative:

1. Compress the log file using an operating system compression utility:

   ```
   % tar czf logfile.tar logfile.txt
   ```

2. Append the compressed log file to an e-mail message to your Technical Support representative.

   Note: Always include the compressed log file as an attachment to the message. *Do not cut and paste the log file* into the body of the e-mail.

3. If the compressed log file is too large to attach to an e-mail message, you can use the BEA Customer Support FTP site.

Addressing Common Problems

The following sections provide solutions to common cluster-related problems. They also provide information for how to diagnose non-specific problems, such as poor cluster performance.

Server Fails to Join a Cluster

There are several reasons why a WebLogic Server does not join a cluster on startup, including general network availability and WebLogic-specific configuration problems. Use this checklist to check your configuration and startup process.

1. Check your command-line parameters for typos, misspellings, etc.

2. Verify that there are no physical problems with your network connection. Network connections can be verified using the `dbping` utilities discussed in Testing Connections.

3. Verify that no other application is using the cluster multicast address.

4. Run the utils.MulticastTest utility to verify that multicast is working.

Other items which require troubleshooting include general configuration errors and communications errors, such as:

- **Incompatible version numbers**. All WebLogic Servers in the cluster must be the same version. If a server attempts to join a cluster with a WebLogic Server whose version does not match the other servers in the cluster, an error message will be generated.

- **Unable to find a license for clustering**. Your WebLogic license does not include the clustering feature. Contact your sales representative.

- **Unable to send service announcement**. This could indicate a general network problem, or a misconfigured DNS. Clustered servers communicate among themselves over multicast and must share the same (exclusive) multicast address.

- **Cannot set default clusterAddress properties value**. This could mean that another server with the same IP address has already joined the cluster. Check to make sure you do not have duplicate IP addresses assigned to multiple machines.

- **Unable to create a multicast socket for clustering, Multicast socket send error, or Multicast socket receive error**. These communications errors are most likely caused by an incorrect or bad multicast address.

  Note that each operating system has specific configuration requirements for configuring multicast; you should check your operating system documentation for help in correcting this error.

A The WebLogic Cluster API

The following section describes the WebLogic Cluster API including:

■ How to Use the API

How to Use the API

The WebLogic Cluster public API is contained in a single interface,
`weblogic.rmi.cluster.CallRouter`.

```
Class java.lang.Object
   Interface weblogic.rmi.cluster.CallRouter
      (extends java.io.Serializable)
```

A class implementing this interface must be provided to the RMI compiler (`rmic`) to
enable parameter-based routing. Run `rmic` on the service implementation using these
options (to be entered on one line):

```
$ java weblogic.rmic -clusterable -callRouter
   <callRouterClass> <remoteObjectClass>
```

The call router is called by the clusterable stub each time a remote method is invoked.
The router is responsible for returning the name of the server to which the call should
be routed.

Each server in the cluster is uniquely identified by its name as defined with the WebLogic Server Administration Console. These are the names that the method router must use for identifying servers.

Example: Consider the `ExampleImpl` class which implements a remote interface Example, with one method `foo`:

```
public class ExampleImpl implements Example {
  public void foo(String arg) { return arg; }
}
```

This `CallRouter` implementation `ExampleRouter` ensures that all `foo` calls with 'arg' < "n" go to server1 (or server3 if server1 is unreachable) and that all calls with 'arg' >= "n" go to server2 (or server3 if server2 is unreachable).

```
public class ExampleRouter implements CallRouter {
  private static final String[] aToM = { "server1", "server3" };
  private static final String[] nToZ = { "server2", "server3" };

  public String[] getServerList(Method m, Object[] params) {
    if (m.GetName().equals("foo")) {
      if (((String)params[0]).charAt(0) < 'n') {
        return aToM;
      } else {
        return nToZ;
      }
    } else {
      return null;
    }
  }
}
```

This `rmic` call associates the `ExampleRouter` with `ExampleImpl` to enable parameter-based routing:

```
$ rmic -clusterable -callRouter ExampleRouter ExampleImpl
```

Custom Call Routing and Collocation Optimization

If a replica is available on the same server instance as the object calling it, the call will not be load-balanced, because it is more efficient to use the local replica. For more information, see "Optimization for Collocated Objects" on page 4-8.

B Configuring BIG-IP™ Hardware with Clusters

This topic contains the following sections:

- Overview
- Using URL Rewriting With BIG-IP and WebLogic Server
- Using Session Persistence with BIG-IP and WebLogic Server

Overview

This section describes how to configure an F5 BIG-IP controller to operate with a WebLogic Server cluster. It assumes that you have some familiarity with BIG-IP configuration tasks.

This section provides some step-by-step instructions for configuring BIG-IP. However, if you require detailed setup and administration instructions, refer to your F5 product documentation.

For information about how WebLogic Server works with external load balancers, see "Accessing Clustered Servlets and JSPs with Load Balancing Hardware" on page 3-10.

Using URL Rewriting With BIG-IP and WebLogic Server

Use of URL rewriting with BIG-IP and WebLogic Server instances requires BIG-IP version 4.5 or higher, configured for `Rewrite cookie persistence`. Failover may not succeed if BIG-IP is set for other persistence settings.

For instructions to configure WebLogic Server for URL rewriting, see "Using URL Rewriting" in *Assembling and Configuring Web Applications*.

Using Session Persistence with BIG-IP and WebLogic Server

If your cluster uses in-memory replication for client session states, you must configure BIG-IP to use the Insert mode for cookies. Insert mode insures that the original WebLogic Server cookie is not overwritten, and can be used in the event that a client fails to connect to its primary WebLogic Server.

To configure Insert mode for BIG-IP cookies:

1. Open the BIG-IP configuration utility.

2. Select the Pools option from the navigation pane.

3. Select the an available pool to configure.

4. Select the Persistence tab.

5. Select Active HTTP Cookie to begin configuring cookies.

6. Choose Insert mode from the list of methods.

7. Enter the timeout value for the cookie. The timeout value specifies how long the inserted cookie remains on the client before expiring. Note that the timeout value does not affect the WebLogic Server session cookie—it affects only the inserted BIG-IP cookie.

   To load balance requests on a round-robin basis, set the timeout value to zero— this ensures that multiple requests from the same client are directed to the same managed server, and that a request from a different client is routed to another managed server in the cluster, in round-robin fashion.

   When the timeout value is set to a value greater than zero, the load balancer sends *all* requests from *all* clients to the *same* managed server in the WebLogic Server cluster for the duration of the timeout period—in other words, requests from different clients will not be load balanced for the duration of the timeout.

8. Apply your changes and exit the utility.