BEA WebLogic Server®

Using WebLogic Server Clusters

BETA

Version 9.0 BETA
Revised: December 15, 2004

BETA

Copyright

Copyright © 2004-2005 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, BEA WebLogic Server, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic JRockit, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

BETA

Contents

1. Introduction and Roadmap

1. Understanding WebLogic Server Clustering

BETA

2. Communications in a Cluster

3. Understanding Cluster Configuration and Application Deployment

BETA

4. Load Balancing in a Cluster

BETA

5. Failover and Replication in a Cluster

BETA

6. Cluster Architectures

BETA

BETA

7. Setting up WebLogic Clusters

BETA

8. Clustering Best Practices

BETA

9. Troubleshooting Common Problems

BETA

A. The WebLogic Cluster API

B. Configuring BIG-IP™ Hardware with Clusters

BETA

CHAPTER 1

Introduction and Roadmap

BETA

This section describes the contents and organization of this guide—*Using WebLogic Server® Clusters*.

Document Scope and Audience

This document is written for application developers and administrators who are developing or deploying Web-based applications on one or more clusters. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server clusteres for a particular application.

The topics in this document are primarly relevant to planning, implementing, and supporting a production environment that includes WebLogic Server clusters. Key guidelines for software engineers who design or develop applications that will run on a WebLogic Server cluster are also addressed.

It is assumed that the reader is familiar with J2EE, HTTP, HTML coding, and Java programming (servlets, JSP, or EJB development).

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," describes the organization of this guide.
- Chapter 1, "Understanding WebLogic Server Clustering"
- Chapter 2, "Communications in a Cluster"
- Chapter 3, "Understanding Cluster Configuration and Application Deployment"
- Chapter 4, "Load Balancing in a Cluster"
- Chapter 5, "Failover and Replication in a Cluster"
- Chapter 6, "Cluster Architectures"
- Chapter 7, "Setting up WebLogic Clusters"
- Chapter 8, "Clustering Best Practices"
- Chapter 9, "Troubleshooting Common Problems"
- Appendix A, "The WebLogic Cluster API"
- Appendix B, "Configuring BIG-IP™ Hardware with Clusters".

Related Documentation

- "Understanding Enterprise JavaBeans (EJBs)" in *Programming WebLogic Enterprise JavaBeans*
- "Creating and Configuring Web Applications" in *Developing Web Applications, Servlets, and JSPs for WebLogic Server*

Samples and Tutorials for Clusters

New and Changed Clustering Features in This Release

The new features in WebLogic Server 9.0 that relate to clustering are describe in the following sections.

BETA

Server Migration

WebLogic Server 9.0 supports automatic and manual migration of a clustered server instance from one machine to another. A Managed Server that can be migrated is referred to as a *migratable server*. This feature is designed for environments with requirements for high availability.The server migration capability is useful for

- Ensuring uninterrupted availability of *singleton services*—services that must run on only a single server instance at any given time, such as JMS and the JTA transaction recovery system, when the hosting server instance fails. A Managed Server configured for automatic migration will be automatically migrated to another machine in the even of failure.
- Easing the process of relocating a Managed Server, and all the services it hosts, as part of a planned system administration process. An administrator can initiate the migration of a Managed Server from the Administration Console or command line.

The server migration process relocates a Managed Server in its entirety—including IP addresses and hosted applications—to on of a predefined set of available host machines.

Note: Previous releases of WebLogic Server provided the ability to migrate an individual singleton service from one server instance to another. This capability is still supported in WebLogic Server 9.0.

Site Failover

Dynamic Generation of Cluster Address

In previous releases, it was necessary to explicitly define a cluster address when configuring a cluster.

In WebLogic Server 9.0 you can still explicilty define a cluster address, but if you do not, WebLogic dynamically generates the cluster address for each request. For more information, see "Cluster Address" on page 7-6.

BETA

BETA

CHAPTER 1

Understanding WebLogic Server Clustering

BETA

This section is a brief introduction to WebLogic Server clusters. It contains the following information:

What Is a WebLogic Server Cluster?

A WebLogic Server cluster consists of multiple WebLogic Server server instances running simultaneously and working together to provide increased scalability and reliability. A cluster appears to clients to be a single WebLogic Server instance. The server instances that constitute a cluster can run on the same machine, or be located on different machines. You can increase a cluster's capacity by adding additional server instances to the cluster on an existing machine, or you can add machines to the cluster to host the incremental server instances. Each server instance in a cluster must run the same version of WebLogic Server.

How Does a Cluster Relate to a Domain?

A cluster is part of a particular WebLogic Server *domain*.

A domain is an interrelated set of WebLogic Server resources that are managed as a unit. A domain includes one or more WebLogic Server instances, which can be clustered, non-clustered, or a combination of clustered and non-clustered instances. A domain can include multiple clusters. A domain also contains the application components deployed in the domain, and the resources and services required by those application components and the server instances in the domain. Examples of the resources and services used by applications and server instances include machine definitions, optional network channels, connectors, and startup classes.

You can use a variety of criteria for organizing WebLogic Server instances into domains. For instance, you might choose to allocate resources to multiple domains based on logical divisions of the hosted application, geographical considerations, or the number or complexity of the resources under management. For additional information about domains see Understanding Domain Configuration.

In each domain, one WebLogic Server instance acts as the Administration Server—the server instance which configures, manages, and monitors all other server instances and resources in the domain. Each Administration Server manages one domain only. If a domain contains multiple clusters, each cluster in the domain has the same Administration Server.

All server instances in a cluster must reside in the same domain; you cannot "split" a cluster over multiple domains. Similarly, you cannot share a configured resource or subsystem between domains. For example, if you create a JDBC connection pool in one domain, you cannot use it with a server instance or cluster in another domain. (Instead, you must create a similar connection pool in the second domain.)

Clustered WebLogic Server instances behave similarly to non-clustered instances, except that they provide failover and load balancing. The process and tools used to configure clustered WebLogic Server instances are the same as those used to configure non-clustered instances. However, to achieve the load balancing and failover benefits that clustering enables, you must adhere to certain guidelines for cluster configuration.

To understand how the failover and load balancing mechanisms used in WebLogic Server relate to particular configuration options see "Load Balancing in a Cluster" on page 4-1, and "Failover and Replication in a Cluster" on page 5-1.

Detailed configuration recommendations are included throughout the instructions in "Setting up WebLogic Clusters" on page 7-1.

BETA

What Are the Benefits of Clustering?

A WebLogic Server cluster provides these benefits:

- Scalability

  The capacity of an application deployed on a WebLogic Server cluster can be increased dynamically to meet demand. You can add server instances to a cluster without interruption of service—the application continues to run without impact to clients and end users.

- High-Availability

  In a WebLogic Server cluster, application processing can continue when a server instance fails. You "cluster" application components by deploying them on multiple server instances in the cluster—so, if a server instance on which a component is running fails, another server instance on which that component is deployed can continue application processing.

The choice to cluster WebLogic Server instances is transparent to application developers and clients. However, understanding the technical infrastructure that enables clustering will help programmers and administrators maximize the scalability and availability of their applications.

What Are the Key Capabilities of a Cluster?

This section defines, in non-technical terms, the key clustering capabilities that enable scalability and high availability.

- Application Failover

  Simply put, failover means that when an application component (typically referred to as an "object" in the following sections) doing a particular "job"—some set of processing tasks—becomes unavailable for any reason, a copy of the failed object finishes the job.

  For the new object to be able to take over for the failed object:

  – There must be a copy of the failed object available to take over the job.

  – There must be information, available to other objects and the program that manages failover, defining the location and operational status of all objects—so that it can be determined that the first object failed before finishing its job.

  – There must be information, available to other objects and the program that manages failover, about the progress of jobs in process—so that an object taking over an interrupted job knows how much of the job was completed before the first object failed, for example, what data has been changed, and what steps in the process were completed.

BETA

WebLogic Server uses standards-based communication techniques and facilities—multicast, IP sockets, and the Java Naming and Directory Interface *(JNDI)*—to share and maintain information about the availability of objects in a cluster. These techniques allow WebLogic Server to determine that an object stopped before finishing its job, and where there is a copy of the object to complete the job that was interrupted.

Information about what has been done on a job is called *state*. WebLogic Server maintains information about state using techniques called *session replication* and *replica-aware stubs*. When a particular object unexpectedly stops doing its job, replication techniques enable a copy of the object pick up where the failed object stopped, and finish the job.

- Site Failover
- Server Migration
- WebLogic Server 9.0 supports automatic and manual migration of a clustered server instance from one machine to another. A Managed Server that can be migrated is referred to as a *migratable server.* This feature is designed for environments with requirements for high availability. The server migration capability is useful for
  - Ensuring uninterrupted availability of *singleton services*—services that must run on only a single server instance at any given time, such as JMS and the JTA transaction recovery system, when the hosting server instance fails. A Managed Server configured for automatic migration will be automatically migrated to another machine in the even of failure.
  - Easing the process of relocating a Managed Server, and all the services it hosts, as part of a planned system administration process. An administrator can initiate the migration of a Managed Server from the Administration Console or command line.

  The server migration process relocates a Managed Server in its entirety—including IP addresses and hosted applications—to on of a predefined set of available host machines.
- Load Balancing

  Load balancing is the even distribution of jobs and associated communications across the computing and networking resources in your environment. For load balancing to occur:
  - There must be multiple copies of an object that can do a particular job.
  - Information about the location and operational status of all objects must be available.

  WebLogic Server allows objects to be clustered—deployed on multiple server instances—so that there are alternative objects to do the same job. WebLogic Server shares and maintains the availability and location of deployed objects using multicast, IP sockets, and JNDI.

BETA

A detailed discussion of how communications and replication techniques are employed by WebLogic Server is provided in "Communications in a Cluster" on page 2-1.

What Types of Objects Can Be Clustered?

A clustered application or application component is one that is available on multiple WebLogic Server instances in a cluster. If an object is clustered, failover and load balancing for that object is available. Deploy objects homogeneously—to every server instance in your cluster—to simplify cluster administration, maintenance, and troubleshooting.

Web applications can consist of different types of objects, including Enterprise Java Beans (EJBs), servlets, and Java Server Pages (JSPs). Each object type has a unique set of behaviors related to control, invocation, and how it functions within an application. For this reason, the methods that WebLogic Server uses to support clustering—and hence to provide load balancing and failover—can vary for different types of objects. The following types of objects can be clustered in a WebLogic Server deployment:

- Servlets
- JSPs
- EJBs
- Remote Method Invocation (RMI) objects
- Java Messaging Service (JMS) destinations
- Java Database Connectivity (JDBC) connections

Different object types can have certain behaviors in common. When this is the case, the clustering support and implementation considerations for those similar object types may be same. In the sections that follow, explanations and instructions for the following types of objects are generally combined:

- Servlets and JSPs
- EJBs and RMI objects

The sections that follow briefly describe the clustering, failover, and load balancing support that WebLogic Server provides for different types of objects.

BETA

Servlets and JSPs

WebLogic Server provides clustering support for servlets and JSPs by replicating the HTTP session state of clients that access clustered servlets and JSPs. WebLogic Server can maintain HTTP session states in memory, a filesystem, or a database.

To enable automatic failover of servlets and JSPs, session state must persist in memory. For information about how failover works for servlets and JSPs, and for related requirements and programming considerations, see "HTTP Session State Replication" on page 5-3.

You can balance the servlet and JSP load across a cluster using a WebLogic Server proxy plug-in or external load balancing hardware. WebLogic Server proxy plug-ins perform round robin load balancing. External load balancers typically support a variety of session load balancing mechanisms. For more information, see "Load Balancing for Servlets and JSPs" on page 4-1.

EJBs and RMI Objects

Load balancing and failover for EJBs and RMI objects is handled using *replica-aware stubs*, which can locate instances of the object throughout the cluster. Replica-aware stubs are created for EJBs and RMI objects as a result of the object compilation process. EJBs and RMI objects are deployed homogeneously—to all the server instances in the cluster.

Failover for EJBs and RMI objects 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. To understand failover support for different types of objects, see "Replication and Failover for EJBs and RMIs" on page 5-13.

WebLogic Server clusters support multiple algorithms for load balancing clustered EJBs and RMI objects: round-robin, weight-based, random, round-robin-affinity, weight-based-affinity, and random-affinity. By default, a WebLogic Server cluster will use the round-robin method. You can configure a cluster to use one of the other methods using the Administration Console. The method you select is maintained within the replica-aware stub obtained for clustered objects. For details, see "Load Balancing for EJBs and RMI Objects" on page 4-4.

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.

BETA

- 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 "Creating and Deploying JDBC Components—Connection Pools, MultiPools, and Data Sources" in the *Administration Console Online Help.*

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.

BETA

- 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 5-34.
- To learn more about how clustered multipools enable load balancing of connections, see "Load Balancing for JDBC Connections" on page 4-16.
- For instructions on configuring clustered JDBC objects, see "Configure Clustered JDBC" on page 7-22.

JMS and Clustering

The WebLogic Java Messaging Service (JMS) architecture implements clustering of multiple JMS servers by supporting cluster-wide, transparent access to destinations from any WebLogic Server server instance in the cluster. Although WebLogic Server supports distributing JMS destinations and connection factories throughout a cluster, the same JMS topic or queue is still managed separately by each WebLogic Server instance in the cluster.

Load balancing is supported for JMS. To enable load balancing, you must configure targets for JMS servers. For more information about load balancing and JMS components, see "Load Balancing for JMS" on page 4-14. For instructions on setting up clustered JMS, see "Configure

BETA

Migratable Targets for Pinned Services" on page 7-22 and "Deploying, Activating, and Migrating Migratable Services" on page 7-26.

What Types of Objects Cannot Be Clustered?

The following APIs and internal services cannot be clustered in WebLogic Server:

- File services
- Time services
- WebLogic Events (deprecated in WebLogic Server 6.0)
- Workspaces (deprecated in WebLogic Server 6.0)

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.

BETA

BETA

CHAPTER 2

Communications in a Cluster

BETA

WebLogic Server clusters implement two key features: load balancing and failover. The following sections provide information that helps architects and administrators configure a cluster that meets the needs of a particular Web application:

WebLogic Server Communication in a Cluster

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

- IP multicast, which server instances use to broadcast availability of services and heartbeats that indicate continued availability.
- IP sockets, which are the conduits for peer-to-peer communication between clustered server instances.

The way in which WebLogic Server uses IP multicast and socket communication affects the way you 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 broadcasts 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 when messages are "dropped." Because of this limitation, WebLogic Server instances allow 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 communications among server instances in a cluster. This communication includes:

- Cluster-wide JNDI updates—Each WebLogic Server instance in a cluster uses multicast to announce the availability of clustered objects that are deployed or removed locally. Each server instance in the cluster monitors these announcements and updates its local JNDI tree to reflect current deployments of clustered objects. For more details, see "Cluster-Wide JNDI Naming Service" on page 2-9.
- Cluster heartbeats— Each WebLogic Server instance in a cluster uses multicast to broadcast regular "heartbeat" messages that advertise its availability. By monitoring heartbeat messages, server instances in a cluster determine when a server instance has failed. (Clustered server instances also monitor IP sockets as a more immediate method of determining when a server instance has failed.)

Multicast and Cluster Configuration

Because multicast communications control critical functions related to detecting failures and maintaining the cluster-wide JNDI tree (described in "Cluster-Wide JNDI Naming Service" on page 2-9) it is important that neither the cluster configuration nor the network topology interfere with multicast communications. The sections that follow provide guidelines for avoiding problems with multicast communication in a cluster.

If Your Cluster Spans Multiple Subnets in a WAN

In many deployments, clustered server instances reside within a single subnet, ensuring multicast messages are reliably transmitted. However, you may want to distribute a WebLogic Server cluster across multiple subnets in a Wide Area Network (WAN) to increase redundancy, or to distribute clustered server instances over a larger geographical area.

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

BETA

- Full support of IP multicast packet propagation. In other words, all routers and other tunneling technologies must be configured to propagate multicast messages to clustered server instances.
- Network latency low enough to ensure that most multicast messages reach their final destination in 200 to 300 milliseconds.
- Multicast Time-To-Live (TTL) value for the cluster high enough to ensure that routers do not discard multicast packets before they reach their final destination. For instructions on setting the Multicast TTL parameter, see "Configure Multicast Time-To-Live (TTL)" on page 7-32.

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 server instances 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. Treat each WebLogic Server cluster as a logical unit that provides one or more distinct services to clients of a Web application. Do not split this logical unit between different security zones. Furthermore, any technologies that potentially delay or interrupt IP traffic can disrupt a WebLogic Server cluster by generating false failures due to missed heartbeats.

Do Not Share the Cluster Multicast Address with Other Applications

Although multiple WebLogic Server clusters can share a single IP multicast address and port, other applications should not broadcast or subscribe to the multicast address and port used by your cluster or clusters. That is, if the machine or machines that host your cluster also host other applications that use multicast communications, make sure that those applications use a different multicast address and port than the cluster does.

Sharing the cluster multicast address with other applications forces clustered server instances to process unnecessary messages, introducing overhead. Sharing a multicast address may also overload the IP multicast buffer and delay transmission of WebLogic Server heartbeat messages. Such delays can 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.

BETA

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. See "Configure Multicast Buffer Size" on page 7-33.

Peer-to-Peer Communication Using IP Sockets

IP sockets provide a simple, high-performance mechanism for transferring messages and data between two applications. Clustered WebLogic Server instances use IP sockets for:

- Accessing non-clustered objects deployed to another clustered server instance on a different machine.
- Replicating HTTP session states and stateful session EJB states between a primary and secondary server instance.
- Accessing clustered objects that reside on a remote server instance. (This generally occurs only in a multi-tier cluster architecture, such as the one described in "Recommended Multi-Tier Architecture" on page 6-6.)

Note: The use of IP sockets in WebLogic Server 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 instance's host system uses a native or a pure-Java socket reader implementation.
- For systems that use pure-Java socket readers, whether the server instance is configured to use enough socket reader threads.

Pure-Java Versus Native Socket Reader Implementations

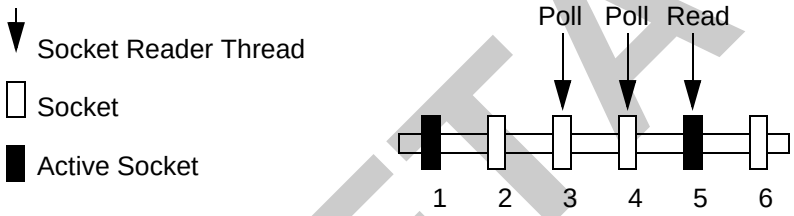
Although the pure-Java implementation of socket reader threads is a reliable and portable method of peer-to-peer communication, it does not provide the best performance for heavy-duty socket usage in a WebLogic Server cluster. With pure-Java socket readers, threads must actively poll all

BETA

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 unnecessary overhead can reduce performance.

The performance issue is magnified when a server instance has more open sockets than it has socket reader threads—each reader thread must poll more than one open socket. When the socket reader encounters an inactive socket, it waits for a timeout before servicing another. During this timeout period, an active socket may go unread while the socket reader polls inactive sockets, as shown in the following figure.

Figure 2-1 Pure-Java Socket Reader Threads Poll Inactive Sockets



BETA

For best socket performance, 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.

Note: Applets cannot use native socket reader implementations, and therefore have limited efficiency in socket communication.

For instructions on how to configure the WebLogic Server host machine to use the native socket reader implementation for your operating system, see “Configure Native IP Sockets Readers on Machines that Host Server Instances” on page 7-31.

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 each

server instance. For best performance, the number of socket reader threads in WebLogic Server should equal the potential maximum number of opened sockets. This configuration avoids the situation in which a reader thread must service multiple sockets, and ensures that socket data is read immediately.

To determine the proper number of reader threads for server instances in your cluster, see the following section, "Determining Potential Socket Usage."

For instructions on how to configure socket reader threads, see "Set the Number of Reader Threads on Machines that Host Server Instances" on page 7-32.

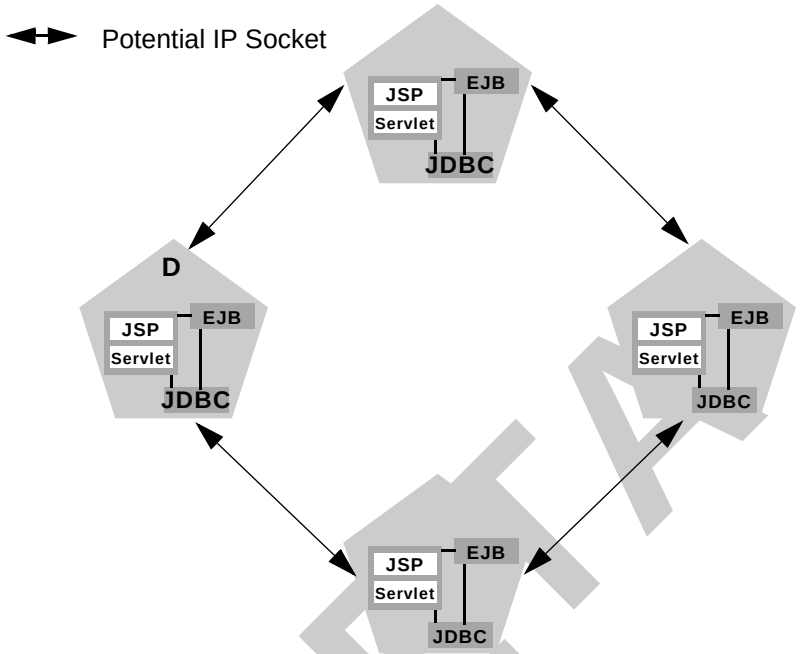
Determining Potential Socket Usage

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 depends on the configuration of your cluster. In practice, clustered systems generally do not open a socket for every other server instance, because objects are deployed homogeneously—to each server instance in the cluster.

If your cluster uses in-memory HTTP session state replication, and you deploy objects homogeneously, each server instance potentially opens a maximum of only two sockets, as shown in the following figure.

BETA

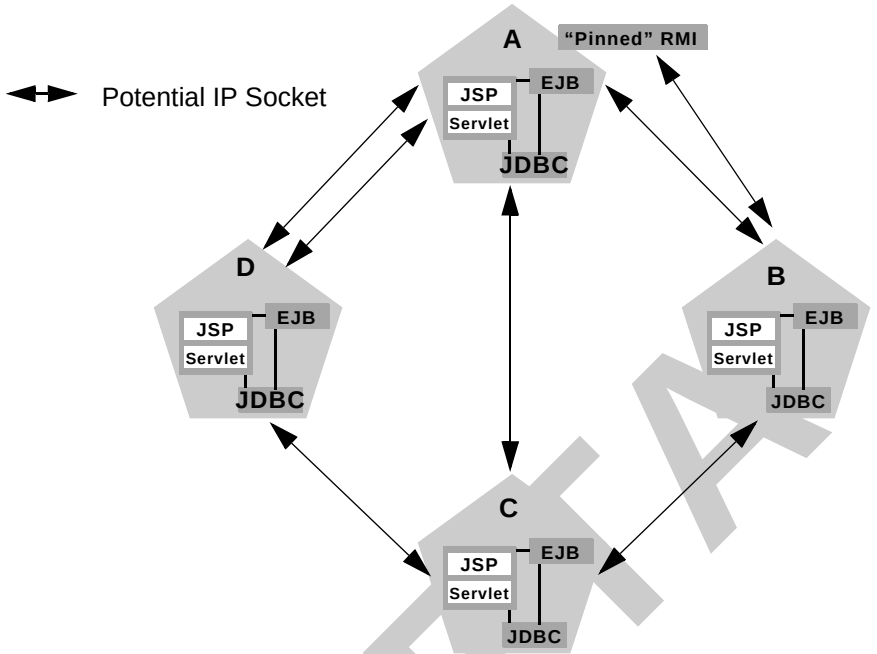
Figure 2-2 Homogeneous Deployment Minimizes Socket Requirements



The two sockets in this example are used to replicate HTTP session states between primary and secondary server instances. Sockets are not required for accessing clustered objects, due to the collocation optimizations that WebLogic Server uses to access those objects. (These optimizations are described in "Optimization for Collocated Objects" on page 4-12.) In this configuration, the default socket reader thread configuration is sufficient.

Deployment of "pinned" services—services that are active on only one server instance at a time—can increase socket usage, because server instances may need to open additional sockets to access the pinned object. (This potential can only be released if a remote server instance actually accesses the pinned object.) The following figure shows the potential effect of deploying a non-clustered RMI object to Server A.

Figure 2-3 Non-Clustered Objects Increase Potential Socket Requirements



In this example, each server instance 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 "Configuration Notes for Multi-Tier Architecture" on page 7-34.

Client Communication via Sockets

Clients of a cluster use the Java implementation of socket reader threads.

In WebLogic Server 8.1, you can configure server affinity load balancing algorithms that reduce the number of IP sockets opened by a Java client application. A client accessing multiple objects on a server instance will use a single socket. If an object fails, the client will failover to a server instance to which it already has an open socket, if possible. In 7.0, under some circumstances, a client might open a socket to each server instance in a cluster.

BETA

For best performance, configure enough socket reader threads in the Java Virtual Machine (JVM) that runs the client. For instructions, see "Set the Number of Reader Threads on Client Machines" on page 7-32.

Cluster-Wide JNDI Naming Service

Clients of a non-clustered WebLogic Server server instance 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 instance offers, organized in a tree structure. A WebLogic Server instance 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 instance 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 instance 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 server instances in the cluster.

Each WebLogic Server instance in a cluster creates and maintains a local copy of the logical cluster-wide JNDI tree. The follow sections describe how the cluster-wide JNDI tree is maintained, and how to avoid naming conflicts that can 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 instance's JNDI entries to other server instances in the cluster, those entries are removed from the cluster if the original instance fails. Also, storing large objects within the JNDI tree can overload multicast traffic and interfere with the normal operation of a cluster.

BETA

How WebLogic Server Creates 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. Creation of a cluster-wide JNDI tree begins with the local JNDI tree bindings of each server instance. As a server instance boots (or as new services are dynamically deployed to a running server instance), the server instance 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.

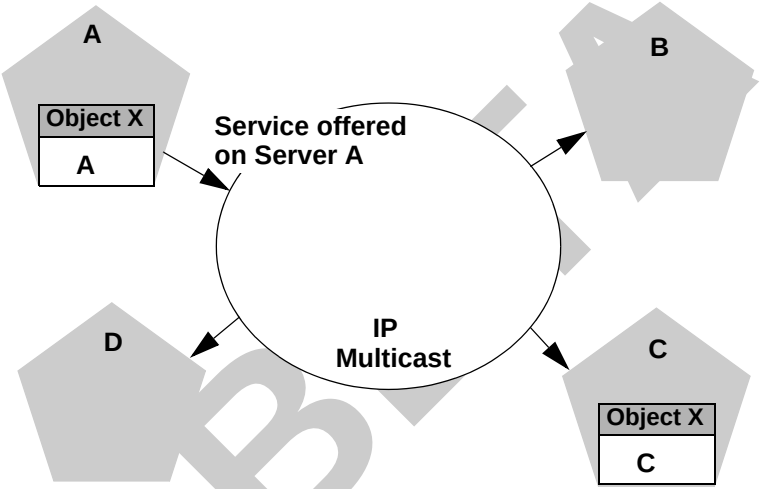
Note: When you start a Managed Server in a cluster, the server instance identifies other running server instances in the cluster by listening for heartbeats, after a warm-up period

specified by the `MemberWarmupTimeoutSeconds` parameter in `ClusterMBean`. The default warm-up period is 30 seconds.

Once the server instance successfully binds a service into the local JNDI tree, additional steps are performed for clustered objects that use replica-aware stubs. After binding the clustered object's implementation into the local JNDI tree, the server instance sends the object's stub to other members of the cluster. Other members of the cluster monitor the multicast address to detect when remote server instances offer new services.

The following figure shows a snapshot of the JNDI binding process.

Figure 2-4 Server A Binds an Object in its JNDI Tree, then Multicasts Object Availability



In the previous figure, 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 X.

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

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

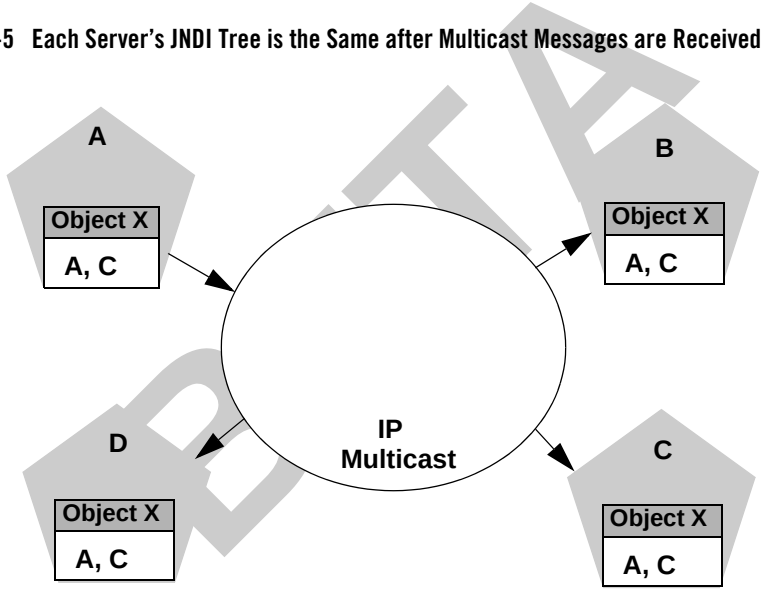
- If the clustered service is not yet bound in the local JNDI tree, the server instance binds a new replica-aware stub into the local tree that indicates the availability of Object X on

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 server instances.

- If the server instance 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 instance 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 instance in the cluster would have identical local JNDI trees that indicate the availability of the object on Servers A and C, as shown below.

Figure 2-5 Each Server's JNDI Tree is the Same after Multicast Messages are Received



Note: In an actual cluster, Object X would be deployed homogeneously, and an implementation which can invoke the object would be available on all four server instances.

How JNDI Naming Conflicts Occur

Simple JNDI naming conflicts occur when a server instance 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 instance 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 instance in the cluster, you cannot deploy a replica-aware version of the same object on another server instance.

If two server instances in a cluster attempt to bind different clustered objects using the same name, both will succeed in binding the object locally. However, each server instance will refuse to bind the other server instance'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 server instances was shut down, or until one of the server instances undeployed the clustered object. This same conflict could also occur if both server instances attempt to deploy a pinned object with the same name.

Deploy Homogeneously to Avoid Cluster-Level JNDI Conflicts

To avoid cluster-level JNDI conflicts, you must homogeneously deploy all replica-aware objects to all WebLogic Server instances in a cluster. 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 server instances, do not replicate the object's bindings across the cluster.

How WebLogic Server Updates the JNDI Tree

When a clustered object is removed (undeployed from a server instance), updates to the JNDI tree are handled similarly to the updates performed when new services are added. The server instance on which the service was undeployed broadcasts a message indicating that it no longer provides the service. Again, other server instances 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 instance 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 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

BETA

implementations of the object. The stub also contains the load balancing logic for distributing the load among its host servers.

For more information about replica-aware stubs for EJBs and RMI classes, see "Replication and Failover for EJBs and RMIs" on page 5-13.

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.*

BETA

BETA

CHAPTER 3

Understanding Cluster Configuration and Application Deployment

BETA

This following sections explain how the information that defines the configuration of a cluster is stored and maintained, and the methods you can use to accomplish configuration tasks:

Note: Much of the information in this section also pertains to the process of configuring a WebLogic domain in which the server instances are not clustered.

Cluster Configuration and config.xml

The `config.xml` file is an XML document that describes the configuration of a WebLogic Server domain. The content and structure of the `config.xml` is defined in the associated Document Type Definition (DTD), `config.dtd`.

`config.xml` consists of a series of XML elements. The Domain element is the top-level element, and all elements in the Domain descend from the Domain element. The Domain element includes child elements, such as the Server, Cluster, and Application elements. These child elements may have children of their own. For example, the Server element includes the child elements WebServer, SSL and Log. The Application element includes the child elements EJBComponent and WebAppComponent.

Each element has one or more configurable attributes. An attribute defined in `config.dtd` has a corresponding attribute in the configuration API. For example, the Server element has a `ListenPort` attribute, and likewise, the `weblogic.management.configuration.ServerMBean` has a `ListenPort` attribute. Configurable attributes are readable and writable, that is, `ServerMBean` has a `getListenPort` and a `setListenPort` method.

To learn more about `config.xml`, see Domain Configuration Files in *Understanding Domain Configuration*.

Role of the Administration Server

The Administration Server is the WebLogic Server instance that configures and manages the WebLogic Server instances in its domain.

A domain can include multiple WebLogic Server clusters and non-clustered WebLogic Server instances. Strictly speaking, a domain could consist of only one WebLogic Server instance—however, in that case that sole server instance would be an Administration Server, because each domain must have exactly one Administration Server.

There are a variety of ways to invoke the services of the Administration Server to accomplish configuration tasks, as described in "Methods of Configuring Clusters" on page 3-8. Whichever method is used, the Administration Server for a cluster must be running when you modify the configuration.

When the Administration Server starts, it loads the `config.xml` for the domain. It looks for `config.xml` in the current directory. Unless you specify another directory when you create a domain, `config.xml` is stored in:

BEA_HOME`/user_projects/`*mydomain*

where *mydomain* is a domain-specific directory, with the same name as the domain.

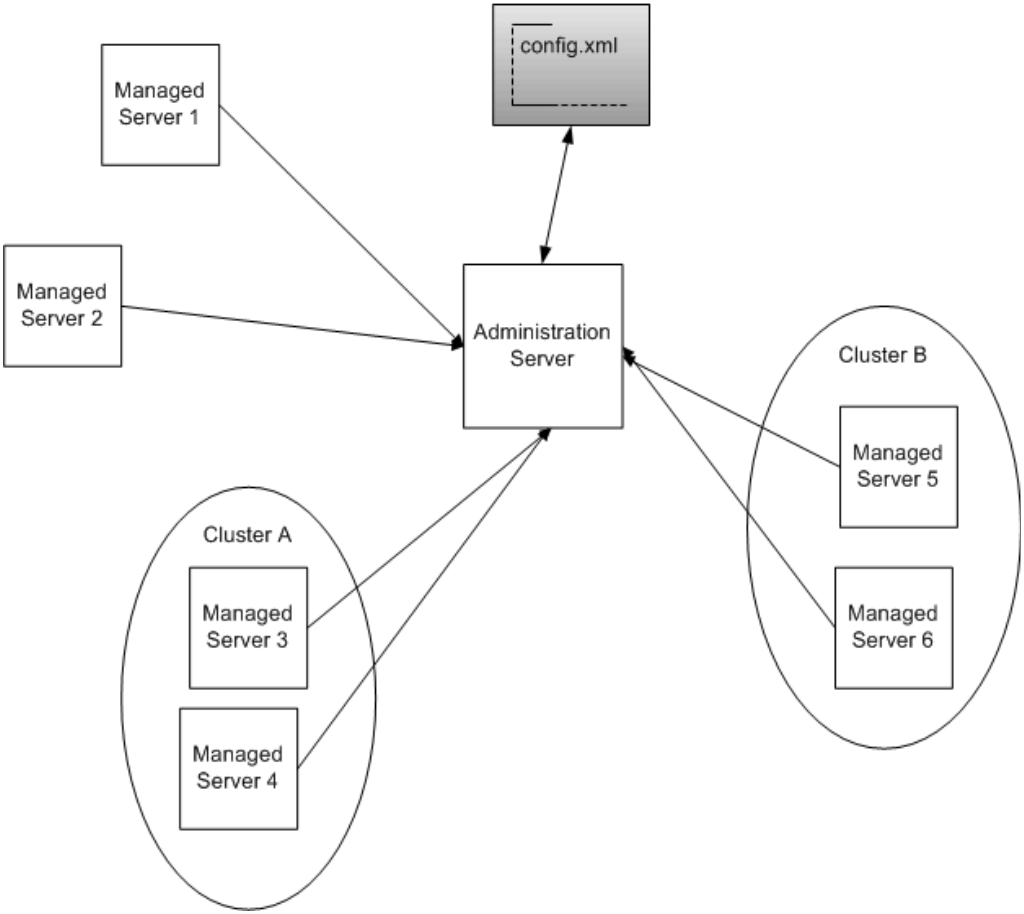
Each time the Administration Server starts successfully, a backup configuration file named `config.xml.booted` is created in the domain directory. In the unlikely event that the `config.xml` file should be corrupted during the lifetime of the server instance, it is possible to revert to this previous configuration.

The following figure shows a typical production environment that contains an Administration Server and multiple WebLogic Servers instances. When you start the server instances in such a domain, the Administration Server is started first. As each additional server instance is started, it contacts the Administration Server for its configuration information. In this way, the

BETA

Administration Server operates as the central control entity for the configuration of the entire domain.

Figure 3-1 WebLogic Server Configuration



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 "Avoiding and Recovering From Server Failure" in *Managing Server Startup and Shutdown*.

How Dynamic Configuration Works

WebLogic Server allows you to change the configuration attributes of domain resources dynamically—while server instances are running. In most cases you do not need to restart the server instance for your changes to take effect. When an attribute is reconfigured, the new value is immediately reflected in both the current run-time value of the attribute and the persistent value stored in `config.xml`.

Not all configuration changes are applied dynamically. For example, if you change a Managed Server's `ListenPort` value, the new port will not be used until the next time you start the Managed Server. The updated value is stored in `config.xml`, but the runtime value is not affected. When the runtime value for a configuration attribute is different than the value in `config.xml`, the Administration Console displays an alert icon—a yellow triangular that contains an exclamation mark (!)—next to the attribute. This icon indicates that the server instance must be restarted for the configuration change to take effect.

The Administration Console validates attribute changes, checking for out-of-range errors and data type mismatch errors, and displays an error message for erroneous entries.

Once the Administration Console has been started, if another process captures the listen port assigned to the Administration Server, you should stop the process that captured the port. If you are not able to remove the process that captured the list port, edit the `config.xml` file to change the `ListenPort` value.

Application Deployment Topics

This section is brief introduction to the application deployment process. For more information about deployment, see *Deploying WebLogic Server Applications*.

For instructions on how to perform common deployment tasks, see "Deploy Applications" on page 7-24.

BETA

Deployment Methods

You can deploy an application to a cluster using following methods:

- WebLogic Server Administration Console

  The Administration Console is a graphical user interface (GUI) to the BEA Administration Service.

- weblogic.Deployer

  The weblogic.Deployer utility is a Java-based deployment tool that provides a command-line interface to the WebLogic Server deployment API.

- WebLogic Scripting Tool

  The WebLogic Scripting Tool (WLST) is a new command-line interface that you can use to automate domain configuration tasks, including application deployment configuration and deployment operations.

These deployment tools are discussed in "Overview of Deployment Tools" in *Deploying WebLogic Server Applications.*

Regardless of the deployment tool you use, when you initiate the deployment process you specify the components to be deployed, and the targets to which they will be deployed—your cluster, or individual server instances within the cluster or domain.

The Administration Server for the domain manages the deployment process, communicating with the Managed Servers in the cluster throughout the process. Each Managed Server downloads components to be deployed, and initiates local deployment tasks. The deployment state is maintained in the relevant MBeans for the component being deployed. For more information, see *Deployment Management API*.

Note: You must package applications before you deploy them to WebLogic Server. For more information, see the packaging topic in "Deploying the Application" in *Developing Applications for WebLogic Server*.

Introduction to Two-Phase Deployment

In WebLogic Server 7.0 and later, applications are deployed in two phases. Before starting, WebLogic Server determines the availability of the Managed Servers in the cluster.

BETA

First Phase of Deployment

During the first phase of deployment, application components are distributed to the target server instances, and the planned deployment is validated to ensure that the application components can be successfully deployed. During this phase, user requests to the application being deployed are not allowed.

Failures encountered during the distribution and validation process will result in the deployment being aborted on all server instances—including those upon which the validation succeeded. Files that have been staged will not be removed; however, container-side changes performed during the preparation will be reverted.

Second Phase of Deployment

After the application components have been distributed to targets and validated, they are fully deployed on the target server instances, and the deployed application is made available to clients.

If a failure occurs during this process, deployment to that server instance will be cancelled. Such a failure on one Managed Server does not prevent successful deployment on other clustered server instances.

If a cluster member fails to deploy an application, it will fail at startup in order to ensure cluster consistency—as any failure of a cluster-deployed application on a Managed Server causes the Managed Server to abort its startup.

Guidelines for Deploying to a Cluster

Ideally, all Managed Servers in a cluster should be running and available during the deployment process. Deploying applications while some members of the cluster are unavailable is not recommended. Before deploying applications to a cluster, ensure, if possible, that all Managed Servers in the cluster are running and reachable by the Administration Server.

Note: In WebLogic Server 8.1, if you deploy an application to a Managed Server that is partitioned at the time of deployment—running but not reachable by the Administration Server—problems accessing the Managed Server can occur when that Managed Server rejoins the cluster. During the synchronization period, while other clustered Managed Servers re-establish communications with the previously partitioned server instance, user requests to the deployed applications and attempts to create secondary sessions on that server instance will fail. The risk of this circumstance occurring can be reduced by setting the `enforceClusterConstraints` flag, as described in "Deploying to Complete Clusters in WebLogic Server" on page 3-7.

BETA

Cluster membership should not change during the deployment process. After initiating deployment, do not:

- add or remove Managed Servers to the target cluster
- shut down Managed Servers in the target cluster

WebLogic Server 8.1 Supports "Relaxed Deployment" Rules

WebLogic Server 7.0 imposed these restrictions on deployment to clusters:

- No partial deployment—In WebLogic Server 7.0, if one or more of the Managed Servers in the cluster are unavailable, the deployment process is terminated, and an error message is generated, indicating that unreachable Managed Servers should be either restarted or removed from the cluster before attempting deployment.
- Pinned services cannot be deployed to multiple Managed Servers in a cluster—If an application is not deployed to the cluster, you can deploy it to one and only one Managed Server in the cluster.

In WebLogic Server 7.0 SP01, these deployment rules were relaxed, allowing user discretion in deployment practices. This behavior remains unchanged in WebLogic Server 8.1.

Deployment to a Partial Cluster is Allowed

By default, WebLogic Server allows deployment to a partial cluster. If one or more of the Managed Servers in the cluster are unavailable, the following message may be displayed:

```
Cannot deploy to the following server(s) because they are
unreachable: "managed_server_n". Make sure that these servers are
currently shut down. Deployment will continue on the remaining
servers in the cluster "clustering". Once deployment has commenced,
do not attempt to start or shutdown any servers until the
application deployment completes.
```

When the unreachable Managed Server becomes available, deployment to that server instance will be initiated. Until the deployment process is completed, the Managed Server may experience failures related to missing or out-of-date classes.

Deploying to Complete Clusters in WebLogic Server

You can ensure that deployment is only performed if all Managed Servers in the cluster are reachable by setting the `enforceClusterConstraints` flag with `weblogic.Deployer`. When `enforceClusterConstraints` is set to "true", deployment will occur in accordance with the

BETA

rules that were enforced in WebLogic 7.0, which are described in "WebLogic Server 8.1 Supports "Relaxed Deployment" Rules" on page 3-7.

Pinned Services can be Deployed to Multiple Managed Servers.

It is possible to target a pinned service to multiple Managed Servers in a cluster. This practice is not recommended. The load-balancing capabilities and scalability of your cluster can be negatively affected by deploying a pinned service to multiple Managed Servers in a cluster. If you target a pined service to multiple Managed Servers, the following message is printed to the server logs:

```
Adding server servername of cluster clustername as a target for
module modulename. This module also includes server servername that
belongs to this cluster as one of its other targets. Having multiple
individual servers a cluster as targets instead of having the entire
cluster as the target can result in non-optimal load balancing and
scalability. Hence this is not usually recommended.
```

Methods of Configuring Clusters

There are several methods for configuring a clusters:

- Domain Configuration Wizard

  The Domain Configuration Wizard is the recommended tool for creating a new domain or cluster. For a list of the tasks you can perform with the wizard, see "Domain Configuration Wizard Capabilities" later in this section.

- WebLogic Server Administration Console

  The Administration Console is a graphical user interface (GUI) to the BEA Administration Service. It allows you to perform a variety of domain configuration and monitoring functions. For a list of the tasks you can perform with the console, see "Administration Console Capabilities" on page 3-9.

- WebLogic Server Application Programming Interface (API)

  You can write a program to modify the configuration attributes, based on the configuration application programming interface (API) provided with WebLogic Server. This method is not recommended for initial cluster implementation.

- WebLogic Server command-line utility.

BETA

You can access the attributes of a domain with the WebLogic Server command-line utility. This utility allows you to create scripts to automate domain management. This method is not recommended for initial cluster implementation.

Domain Configuration Wizard Capabilities

The Domain Configuration Wizard uses pre-configured domain templates to ease the process of creating a domain and its server instances. Using the wizard, you can select a domain template, and then supply key information, such as machine addresses, names, and port numbers for the server instances you wish to created.

Note: The Domain Configuration Wizard can install the appropriate directory structure and scripts for a domain on a Managed Server that is running on a remote machine from the Administration Server. This is helpful if you need to use a Managed Server as a backup Administration Server for a domain.

The wizard prompts you to select one of four typical domain configurations:

- Single Server—domain with a single WebLogic Server instance.
- Administration Server with Managed Servers—domain with an Administration Server, and one or more Managed Servers that are not clustered.
- Administration Server with clustered Managed Servers—domain with an Administration Server, and one or more Managed Servers that are clustered.
- Managed Server (Owning Administrative Configuration)

After you select the desired configuration type, the wizard prompts you to provide relevant details about the domain and its server instances.

For information on how to use the Domain Configuration Wizard, see Creating WebLogic Domains Using the Configuration Wizard.

Administration Console Capabilities

These sections in *Administration Console Online Help* list and describe the cluster-related configuration tasks you can perform using the WebLogic Server Administration Console.

- "Servers"
- "Clusters"
- "Deploying Applications and Modules"

BETA

- “Monitoring a Server”
- “Monitoring a Cluster”

BETA

CHAPTER 4

Load Balancing in a Cluster

BETA

This section describes the load balancing support that a WebLogic Server cluster provides for different types of objects, and related planning and configuration considerations for architects and administrators. It contains the following information:

For information about replication and failover in a cluster, see "Failover and Replication in a Cluster" on page 5-1.

Load Balancing for Servlets and JSPs

Load balancing of servlets and JSPs can be accomplished with the built-in load balancing capabilities of a WebLogic proxy plug-in or with separate load balancing hardware.

Note: In addition to distributing HTTP traffic, external load balancers can distribute initial context requests that come from Java clients over t3 and the default channel. See "Load Balancing for EJBs and RMI Objects" on page 4-4 for a discussion of object-level load balancing in WebLogic Server.

Load Balancing with a 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 on a round-robin basis. This load balancing method is described in "Round Robin Load Balancing" on page 4-4.

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 proxy plug-ins, see "Configure Proxy Plug-Ins" on page 7-14.

How Session Connection and Failover Work with a Proxy Plug-in

For a description of connection and failover for HTTP sessions in a cluster with proxy plug-ins, see "Accessing Clustered Servlets and JSPs Using a Proxy" on page 5-8.

Load Balancing HTTP Sessions with an External Load Balancer

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

Load Balancer Configuration Requirements

If you choose to use load balancing hardware instead of a proxy plug-in, it must support a compatible passive or active cookie persistence mechanism, and SSL persistence.

- Passive Cookie Persistence

  Passive cookie persistence enables WebLogic Server to write a cookie containing session parameter information through the load balancer to the client. For information about the session cookie and how a load balancer uses session parameter data to maintain the relationship between the client and the primary WebLogic Server hosting a HTTP session state, see "Load Balancers and the WebLogic Session Cookie" on page 4-3.

BETA

- Active Cookie Persistence

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

- SSL Persistence

  When SSL persistence is used, the load balancer 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.

Load Balancers and the WebLogic Session Cookie

A load balancer that uses passive cookie persistence can use a string in the WebLogic session cookie to associate a client with the server hosting its primary HTTP session state. The string 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.

The format of a session cookie is:

```
sessionid!primary_server_id!secondary_server_id
```

where:

- `sessionid` is a randomly generated identifier of the HTTP session. The length of the value is configured by the `IDLength` parameter in the `<session-descriptor>` element in the `weblogic.xml` file for an application. By default, the `sessionid` length is 52 bytes.
- `primary_server_id` and `secondary_server_id` are 10 character identifiers of the primary and secondary hosts for the session.

Note: For sessions using non-replicated memory, cookie, JDBC, or file-based session persistence, the `secondary_server_id` is not present. For sessions that use in-memory replication, if the secondary session does not exist, the `secondary_server_id` is "NONE".

For general instructions on configuring load balancers, see "Configuring Load Balancers that Support Passive Cookie Persistence" on page 7-13. Instructions for configuring BIG-IP, see Configuring BIG-IP™ Hardware with Clusters.

BETA

Related Programming Considerations

For programming constraints and recommendations for clustered servlets and JSPs, see "Programming Considerations for Clustered Servlets and JSPs" on page 5-4.

How Session Connection and Failover Works with a Load Balancer

For a description of connection and failover for HTTP sessions in a cluster with load balancing hardware, see "Accessing Clustered Servlets and JSPs with Load Balancing Hardware" on page 5-10.

Load Balancing for EJBs and RMI Objects

This section describes WebLogic Server load balancing algorithms for EJBs and RMI objects.

The load balancing algorithm for an object is maintained in the replica-aware stub obtained for a clustered object.

By default, a WebLogic Server cluster uses round-robin load balancing, described in "Round Robin Load Balancing" on page 4-4. You can configure a different default load balancing method for the cluster by using the Administration Console to set `weblogic.cluster.defaultLoadAlgorithm`. For instructions, see "Configure Load Balancing Method for EJBs and RMIs" on page 7-12. You can also specify the load balancing algorithm for a specific RMI object using the `-loadAlgorithm` option in `rmic`, or with the `home-load-algorithm` or `stateless-bean-load-algorithm` in an EJB's deployment descriptor. A load balancing algorithm that you configure for an object overrides the default load balancing algorithm for the cluster.

In addition to the standard load balancing algorithms, WebLogic Server supports custom parameter-based routing. For more information, see "Parameter-Based Routing for Clustered Objects" on page 4-11.

BETA

Round Robin Load Balancing

WebLogic Server uses the round-robin algorithm as the default load balancing strategy for clustered object stubs when no algorithm is specified. This algorithm is supported for RMI objects and EJBs. It is also the method used by WebLogic proxy plug-ins.

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 Server instances that host the clustered servlet or JSP.

The advantages of the round-robin 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, a slow server can cause requests to "synchronize" on the server, then follow other servers in order for future requests.

Note: WebLogic Server does not always load balance an object's method calls. For more information, see "Optimization for Collocated Objects" on page 4-12.

Weight-Based Load Balancing

This algorithm applies only to EJB and RMI object clustering.

Weight-based load balancing improves on the round-robin algorithm by taking into account a pre-assigned weight for each server. You can use the Server -> Configuration -> Cluster tab in the Administration Console to assign each server in the cluster a numerical weight between 1 and 100, in the Cluster Weight field. This value determines what proportion of the load the server will bear relative to other servers. If all servers have 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, carefully determine the relative weights to assign to each server instance. Factors to consider 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. For related information see "Cluster-Wide JNDI Naming Service" on page 2-9.

Notes: WebLogic Server does not always load balance an object's method calls. For more information, see "Optimization for Collocated Objects" on page 4-12.

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

Random Load Balancing

The random method of load balancing applies only to EJB and RMI object clustering.

BETA

In random load balancing, requests are routed to servers at random. Random load balancing is recommended only for homogeneous cluster deployments, where each server instance runs on a similarly configured machine. A random allocation of requests does not allow for differences in processing power among the machines upon which server instances run. If a machine hosting servers in a cluster has significantly less processing power than other machines in the cluster, random load balancing will give the less powerful machine as many requests as it gives more powerful machines.

Random load balancing distributes requests evenly across server instances in the cluster, increasingly so as the cumulative number of requests increases. Over a small number of requests the load may not be balanced exactly evenly.

Disadvantages of random load balancing include the slight processing overhead incurred by generating a random number for each request, and the possibility that the load may not be evenly balanced over a small number of requests.

Note: WebLogic Server does not always load balance an object's method calls. For more information, see "Optimization for Collocated Objects" on page 4-12.

Server Affinity Load Balancing Algorithms

WebLogic Server provides three load balancing algorithms for RMI objects that provide *server affinity*. Server affinity turns off load balancing for external client connections: instead, the client considers its existing connections to WebLogic server instances when choosing the server instance on which to access an object. If an object is configured for server affinity, the client-side stub attempts to choose a server instance to which it is already connected, and continues to use the same server instance for method calls. All stubs on that client attempt to use that server instance. If the server instance becomes unavailable, the stubs fail over, if possible, to a server instance to which the client is already connected.

The purpose of server affinity is to minimize the number IP sockets opened between external Java clients and server instances in a cluster. WebLogic Server accomplishes this by causing method calls on objects to "stick" to an existing connection, instead of being load balanced among the available server instances. With server affinity algorithms, the less costly server-to-server connections are still load-balanced according to the configured load balancing algorithm—load balancing is disabled only for external client connections.

Server affinity is used in combination with one of the standard load balancing methods: round-robin, weight-based, or random:

BETA

- round-robin-affinity—server affinity governs connections between external Java clients and server instances; round robin load balancing is used for connections between server instances.
- weight-based-affinity—server affinity governs connections between external Java clients and server instances; weight-based load balancing is used for connections between server instances.
- random-affinity—server affinity governs connections between external Java clients and server instances; random load balancing is used for connections between server instances.

Server Affinity and Initial Context

A client can request an initial context from a particular server instance in the cluster, or from the cluster by specifying the cluster address in the URL. The connection process varies, depending on how the context is obtained:

- If the initial context is requested from a specific Managed Server, the context is obtained using a new connection to the specified server instance.
- If the initial context is requested from a the cluster, by default, context requests are load balanced on a round-robin basis among the clustered server instances. To reuse an existing connection between a particular JVM and the cluster, set `ENABLE_SERVER_AFFINITY` to true in the hashtable of `weblogic.jndi.WLContext` properties you specify when obtaining context. (If a connection is not available, a new connection is created.) `ENABLE_SERVER_AFFINITY` is only supported when the context is requested from the cluster address.

Server Affinity and IIOP Client Authentication Using CSIv2

If you use WebLogic Server's Common Secure Interoperability (CSIv2) functionality to support stateful interactions with WebLogic Server's J2EE Application Client ("thin client"), you must use an affinity-based load balancing algorithm to ensure that method calls stick to a server instance. Otherwise, all remote calls will be authenticated. To prevent redundant authentication of stateful CSIv2 clients, use one of the load balancing algorithms described in "Round-Robin Affinity, Weight-Based Affinity, and Random-Affinity" on page 4-7.

Round-Robin Affinity, Weight-Based Affinity, and Random-Affinity

WebLogic Server 8.1 introduces three new load balancing algorithms that provide server affinity:

- round-robin-affinity

BETA

- weight-based-affinity
- random-affinity

Server affinity is supported for all types of RMI objects including JMS objects, all EJB home interfaces, and stateless EJB remote interfaces.

The server affinity algorithms consider existing connections between an external Java client and server instances in balancing the client load among WebLogic server instances. Server affinity:

- turns off load balancing between external Java clients and server instances
- causes method calls from an external Java client to stick to a server instance to which the client has an open connection, assuming that the connection supports the necessary protocol and QOS
- in the case of failure, causes the client to failover to a server instance to which it has an open connection, assuming that the connection supports the necessary protocol and QOS
- does not affect the load balancing performed for server-to-server connections

Server Affinity Examples

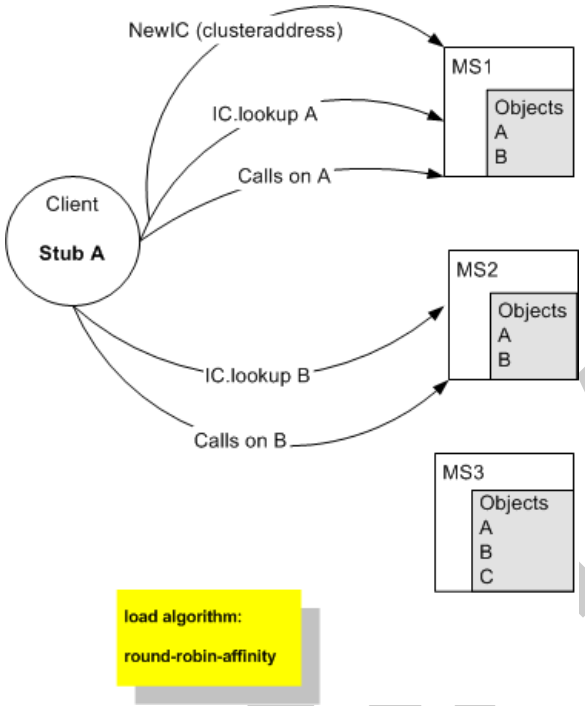
The following examples illustrate the effect of server affinity under a variety of circumstances. In each example, the objects deployed are configured for round-robin-affinity.

Example 1—Context from cluster

In this example, the client obtains context from the cluster. Lookups on the context and object calls stick to a single connection. Requests for new initial context are load balanced on a round-robin basis.

BETA

Figure 4-1 Client Obtains Context From the Cluster

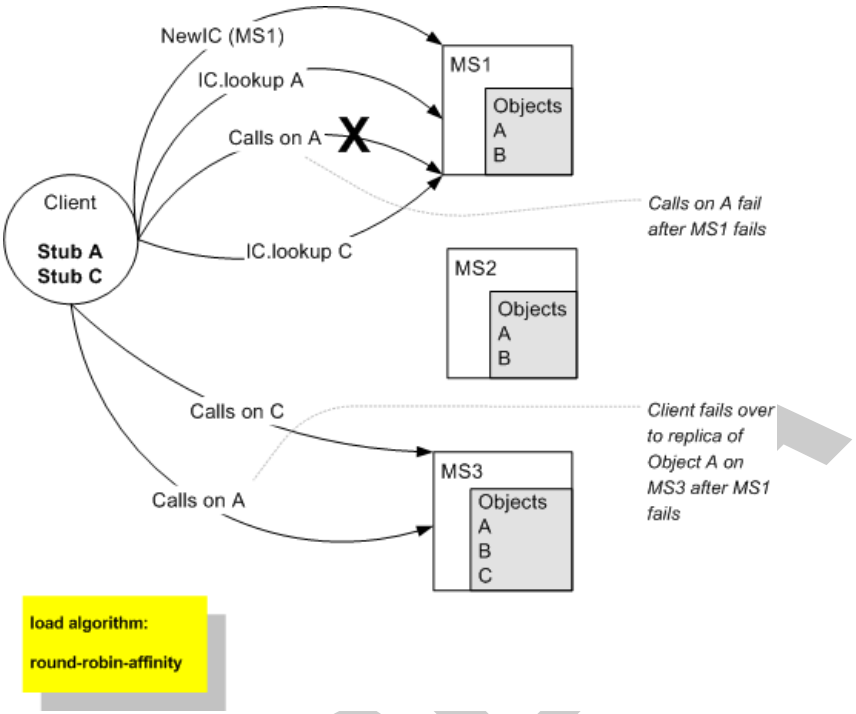


1. Client requests a new initial context from the cluster (`Provider_URL=clusteraddress`) and obtains the context from MS1.
2. Client does a lookup on the context for Object A. The lookup goes to MS1.
3. Client issues a call to Object A. The call goes to MS1, to which the client is already connected. Additional method calls to Object A stick to MS1.
4. Client requests a new initial context from the cluster (`Provider_URL=clusteraddress`) and obtains the context from MS2.
5. Client does a lookup on the context for Object B. The call goes to MS2, to which the client is already connected. Additional method calls to Object B stick to MS2.

Example 2—Server Affinity and Failover

This example illustrates the effect that server affinity has on object failover. When a Managed Server goes down, the client fails over to another Managed Server to which it has a connection.

Figure 4-2 Server Affinity and Failover

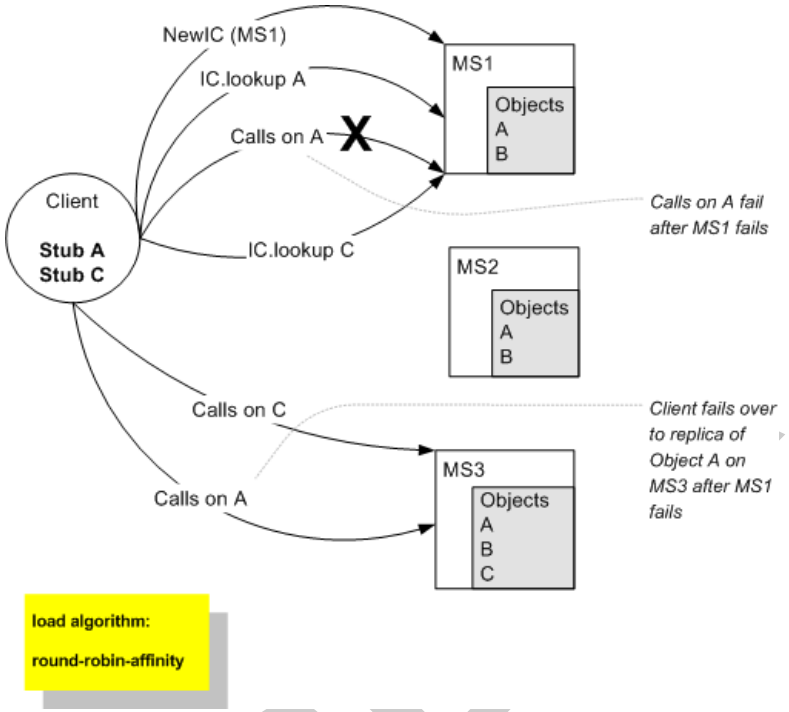


1. Client requests new initial context from MS1.
2. Client does a lookup on the context for Object A. The lookup goes to MS1.
3. Client makes a call to Object A. The call goes to MS1, to which the client is already connected. Additional calls to Object A stick to MS1.
4. The client obtains a stub for Object C, which is pinned to MS3. The client opens a connection to MS3.
5. MS1 fails.
6. Client makes a call to Object A.The client no longer has a connection to MS1. Because the client is connected to MS3, it fails over to a replica of Object A on MS3.

Example 3—Server affinity and server-to-server connections

This example illustrates the fact that server affinity does not affect the connections between server instances.

Figure 4-3 Server Affinity and Server-to-Server Connections



1. A JSP on MS4 obtains a stub for Object B.
2. The JSP selects a replica on MS1. For each method call, the JSP cycles through the Managed Servers upon which Object B is available, on a round-robin basis.

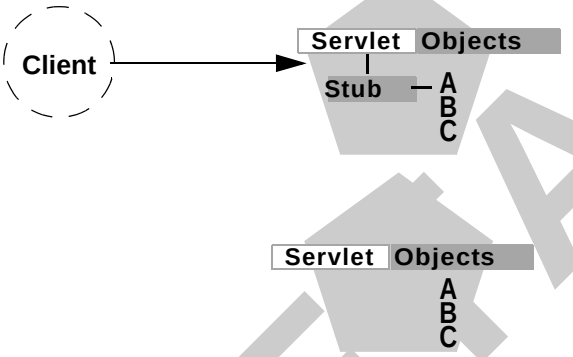
Parameter-Based Routing for Clustered Objects

Parameter-based routing allows you to control load balancing behavior at a lower level. 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. For information about creating custom `CallRouter` classes, see Parameter-Based Routing for Clustered Objects in *Programming WebLogic RMI*.

Optimization for Collocated Objects

WebLogic Server does not always load balance 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 a replica that resides on a remote server. The following figure illustrates this.

Figure 4-4 Collocation Optimization Overrides Load Balancer Logic for Method Call



In this 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 instance, 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. If your Web application is deployed to a single cluster, the collocation optimization overrides any load balancing logic inherent in the replica-aware stub.

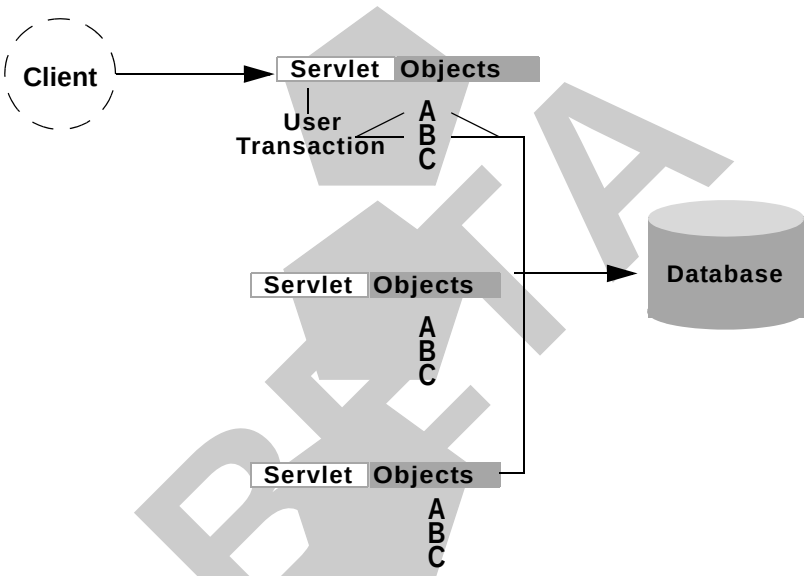
If you require load balancing on each method call to a clustered object, see "Recommended Multi-Tier Architecture" on page 6-6 for information about how to plan your WebLogic Server cluster accordingly.

BEA

Transactional Collocation

As an extension to the basic collocation strategy, WebLogic Server attempts to use collocated 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.

Figure 4-5 Collocation Optimization Extends to Other Objects in Transaction



In this example, a client attaches to he 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" on page 4-12. 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 additional network overhead.

By using collocating clustered objects during a transaction, 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.

Load Balancing for JMS

WebLogic Server JMS supports server affinity for distributed JMS destinations and client connections.

By default, a WebLogic Server cluster uses the round-robin method to load balance objects. To use a load balancing algorithm that provides server affinity for JMS objects, you must configure the desired method for the cluster as a whole. You can configure the load balancing algorithm by using the Administration Console to set `weblogic.cluster.defaultLoadAlgorithm`. For instructions, see "Configure Load Balancing Method for EJBs and RMIs" on page 7-12.

Server Affinity for Distributed JMS Destinations

Server affinity is supported for JMS applications that use the distributed destination feature; this feature is not supported for standalone destinations. If you configure server affinity for JMS connection factories, a server instance that is load balancing consumers or producers across multiple members of a distributed destination will first attempt to load balance across any destination members that are also running on the same server instance.

For detailed information on how the JMS connection factory's Server Affinity Enabled option affects the load balancing preferences for distributed destination members, see "How Distributed Destination Load Balancing Is Affected When Using the Server Affinity Enabled Attribute" in *Programming WebLogic JMS*.

For instructions to configure server affinity for distributed destinations, see "Tuning Distributed Destinations" in *Administration Console Online Help*.

Initial Context Affinity and Server Affinity for Client Connections

A system administrator can establish load balancing of JMS destinations across multiple servers in a 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. During the configuration phase, the system administrator

BETA

enables load balancing by specifying targets for JMS servers. For instructions on setting up targets, see "Configure Migratable Targets for Pinned Services" on page 7-22. For instructions on deploying a JMS server to a migratable target, see "Deploying, Activating, and Migrating Migratable Services" on page 7-26.

A system administrator can 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. Connection factories are described in more detail in "Connection Factory" in *Programming WebLogic JMS*.

The application uses the Java Naming and Directory Interface (JNDI) to look up a connection factory and create a connection to establish communication with a JMS server. Each JMS server handles requests for a set of destinations. Requests for destinations not handled by a JMS server are forwarded to the appropriate server.

WebLogic Server 8.1 provides server affinity for client connections. If an application has a connection to a given server instance, JMS will attempt to establish new JMS connections to the same server instance.

When creating a connection, JMS will try first to achieve initial context affinity. It will attempt to connect to the same server or servers to which a client connected for its initial context, assuming that the server instance is configured for that connection factory. For example, if the connection factory is configured for servers A and B, but the client has an InitialContext on server C, then the connection factory will not establish the new connection with A, but will choose between servers B and C.

If a connection factory cannot achieve initial context affinity, it will try to provide affinity to a server to which the client is already connected. For instance, assume the client has an InitialContext on server A and some other type of connection to server B. If the client then uses a connection factory configured for servers B and C it will not achieve initial context affinity. The connection factory will instead attempt to achieve server affinity by trying to create a connection to server B, to which it already has a connection, rather than server C.

If a connection factory cannot provide either initial context affinity or server affinity, then the connection factory is free to make a connection wherever possible. For instance, assume a client has an initial context on server A, no other connections and a connection factory configured for servers B and C. The connection factory is unable to provide any affinity and is free to attempt new connections to either server B or C.

Note: In the last case, if the client attempts to make a second connection using the same connection factory, it will go to the same server as it did on the first attempt. That is, if it

BETA

chose server B for the first connection, when the second connection is made, the client will have a connection to server B and the server affinity rule will be enforced.

Load Balancing for 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 5-34, 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 on clustering JDBC objects, see "Configure Clustered JDBC" on page 7-22.

BETA

CHAPTER 5

Failover and Replication in a Cluster

BETA

In order for a cluster to provide high availability it must be able to recover from service failures. The following sections describe how WebLogic Server detect failures in a cluster, and provides an overview of how failover is accomplished for different types of 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 Server instances 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 heartbeat message 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.

Note: For more information about how WebLogic Server uses IP sockets and multicast communications see "WebLogic Server Communication in a Cluster" on page 2-1.

Replication and Failover for Servlets and JSPs

In clusters that utilize Web servers with WebLogic proxy plug-ins, the proxy plug-in handles failover transparently to the client. If a 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.

BETA

HTTP Session State Replication

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

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 *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 a collection of Web servers with identically configured WebLogic proxy plug-ins, or load balancing hardware.

Supported Server and Proxy Software

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.

In-memory replication for HTTP session states is supported by the following Web server and proxy software:

- 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 proxy plug-ins, see "Configure Proxy Plug-Ins" on page 7-14.

Load Balancer Requirements

If you choose to use load balancing hardware instead of a proxy plug-in, it must support a compatible passive or active cookie persistence mechanism, and SSL persistence. For details on

BETA

these requirements, see "Load Balancer Configuration Requirements" on page 4-2. For instructions on setting up a load balancer, see "Configuring Load Balancers that Support Passive Cookie Persistence" on page 7-13.

Programming Considerations for Clustered Servlets and JSPs

This section highlights key programming constraints and recommendations for servlets and JSPs that you will deploy in a clustered environment.

- Session Data Must Be Serializable

  To support in-memory replication of HTTP session states, all servlet and JSP session data must be serializable.

Note: Serialization is the process of converting a complex data structure, such as a parallel arrangement of data (in which a number of bits are transmitted at a time along parallel channels) into a serial form (in which one bit at a time is transmitted); a serial interface provides this conversion to enable data transmission.

Every field in an object must be serializable or transient in order for the object to be considered 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. Every time a change is made to an object that is in the session, `setAttribute()` should be called to update that object across the cluster.

  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

  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

BETA

objects in the session, test the performance of your servlets to ensure that performance is acceptable.

- Control Frame Access to Session Data

  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, carefully plan how you access 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 that hosts the primary session state. You can further control where secondary states are placed using *replication groups*. A replication group is a preferred list of clustered servers to be used for storing session state replicas.

Using the WebLogic Server 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 the same machine. For example, you would assign the same machine name to all server instances that run on the same machine, or the same server hardware.

If you do not run multiple WebLogic Server instances on a single machine, you do not need to specify WebLogic Server machine names. Servers without a machine name are treated as though they reside on separate machines. For detailed instructions on setting machine names, see "Configure Machine Names" on page 7-33.

BETA

When you configure a clustered server instance, you can assign the server to a replication group, and a preferred secondary replication group 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.

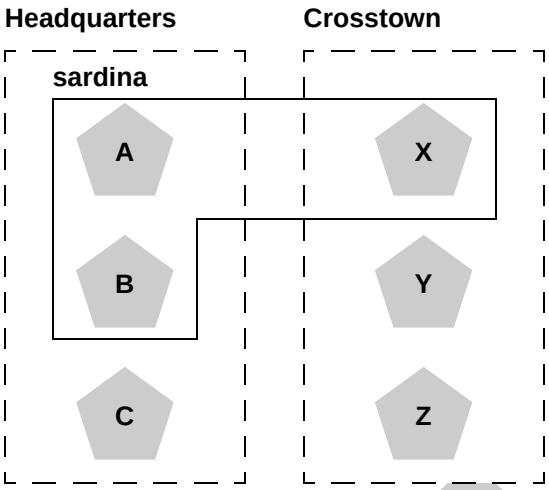
Table 5-1 Ranking Server Instances for Session Replication

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	Yes

Using these rules, 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 for different geographic locations.

BETA

Figure 5-1 Replication Groups for Different 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 configure a server's membership in a replication group, or to assign a server's preferred secondary replication group, follow the instructions in "Configure Replication Groups" on page 7-21.

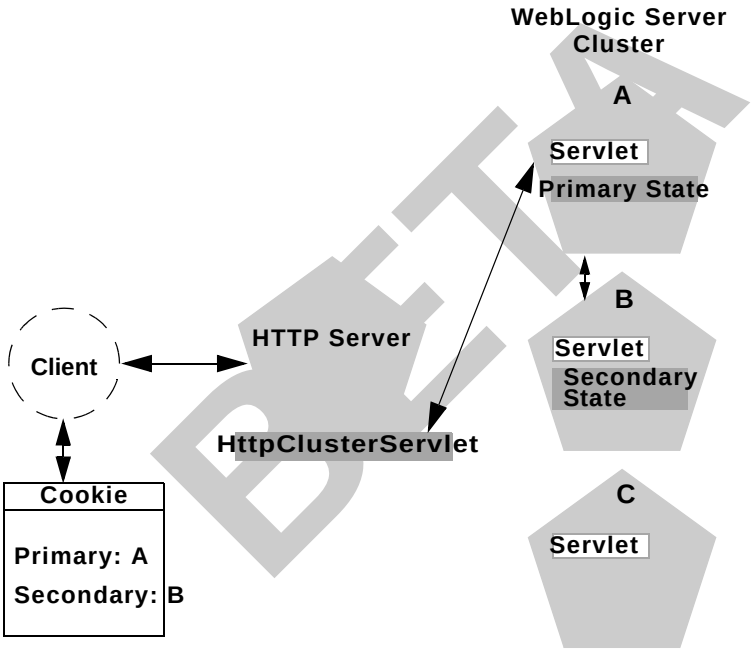
BETA

Accessing Clustered Servlets and JSPs Using a Proxy

This section describes the connection and failover processes for requests that are proxied to clustered servlets and JSPs. For instructions on setting up proxy plug-ins, see "Configure Proxy Plug-Ins" on page 7-14.

The following figure depicts a client accessing a servlet hosted in a cluster. This example uses a single WebLogic Server to serve static HTTP requests only; all servlet requests are forwarded to the WebLogic Server cluster via the `HttpClusterServlet`.

Figure 5-2 Accessing Servlets and JSPs using a Proxy



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`.

Proxy Connection Procedure

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

`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. For instructions on how to enable URL rewriting, see "Using URL Rewriting", in *Assembling and Configuring Web Applications*.

Proxy Failover Procedure

Should the primary server fail, `HttpClusterServlet` uses the client's cookie information to determine the location of the secondary WebLogic Server that hosts the replica of the session state. `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 previous example). 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.

BETA

Accessing Clustered Servlets and JSPs with Load Balancing Hardware

To support direct client access via load balancing hardware, the WebLogic Server replication system allows clients to use secondary session states regardless of the server to which the client fails over. WebLogic Server uses 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.

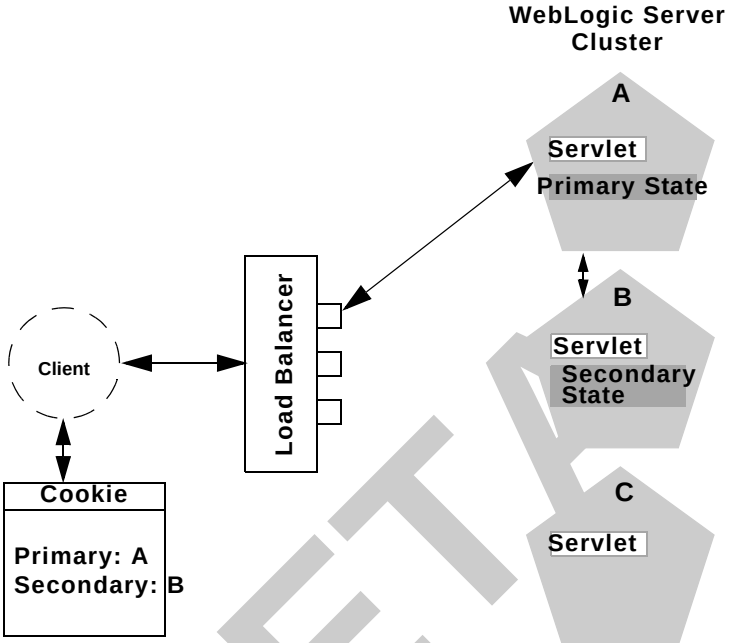
The following sections describe the connection and failover procedure when using HTTP session state replication with load balancing hardware.

Connection with Load Balancing Hardware

The following figure illustrates the connection procedure for a client accessing a cluster through a load balancer.

BETA

Figure 5-3 Connection with Load Balancing Hardware



When the client of a Web application requests a servlet using a public IP address:

1. The load balancer routes the client's connection request to a WebLogic Server cluster in accordance with its configured policies. It directs the request to WebLogic Server A.
2. WebLogic Server A acts as the primary host of the client's servlet session state. It uses the ranking system described in "Using Replication Groups" on page 5-5 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 Server instances 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, as described in "Using URL Rewriting to Track Session Replicas" on page 5-9.

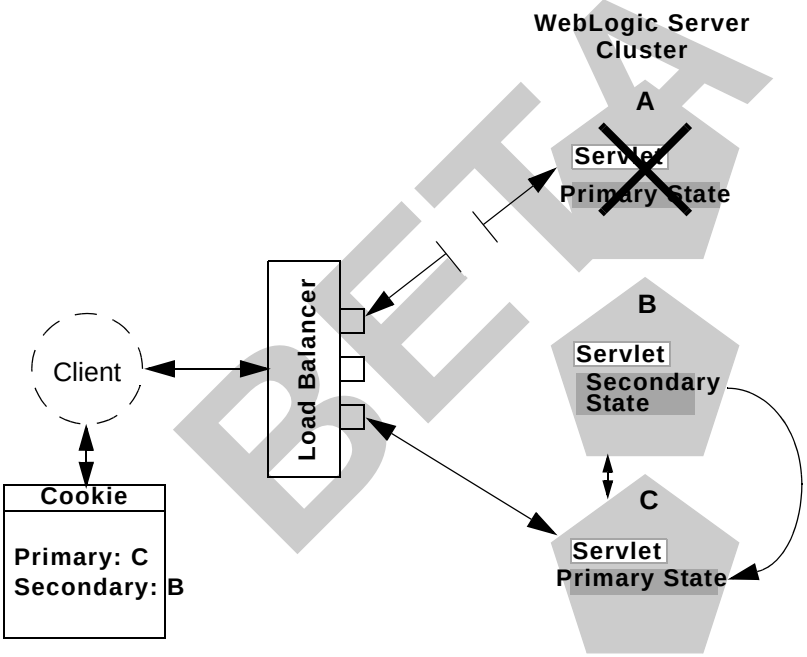
BETA

4. As the client makes additional 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, as illustrated in the following figure.

Figure 5-4 Failover with Load Balancing Hardware



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.

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.

Replication and Failover for EJBs and RMIs

For clustered EJBs and RMIs, 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.

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 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.

Clustering Objects with Replica-Aware Stubs

If an EJB or RMI object is clustered, instances of the object are deployed on all WebLogic Server instances 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 supports object clustering objects in WebLogic Server is the *replica-aware stub*. When you compile an EJB that supports clustering (as defined in its deployment descriptor), `appc` 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," in *Programming WebLogic RMI*.

BETA

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" on page 2-9, clustered WebLogic Server instances have the capability to update the JNDI tree to list all server instances 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. To understand the load balancing algorithms available for RMI objects and EJBs, see "Load Balancing for EJBs and RMI Objects" on page 4-4. 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.

Clustering Support for Different Types of 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 describe clustering support for different types of EJBs.

Clustered EJBHomes

All bean homes interfaces—used to find or create bean instances—can be clustered, by specifying the determined by the `home-is-clusterable` element in `weblogic-ejb-jar.xml`.

Note: Stateless session beans, stateful session beans, and entity beans have home interfaces. Message-driven beans do not.

When a bean is deployed to a cluster, each server binds the bean's home interface to its cluster JNDI tree under the same name. When a client requests the bean's home from the cluster, the server instance that does the look-up returns a `EJBHome` stub that has a reference to the home on each server.

BETA

When the client issues a `create()` or `find()` call, the stub routes selects a server from the replica list in accordance with the load balancing algorithm, and routes the call to the home interface on that server. The selected home interface receives the call, and creates a bean instance on that server instance and executes the call, creating an instance of the bean.

Note: WebLogic Server 8.1 supports new load balancing algorithms that provide server affinity for EJB home interfaces. To understand server affinity and how it affects load balancing and failover, see "Round-Robin Affinity, Weight-Based Affinity, and Random-Affinity" on page 4-7.

Clustered EJBObjects

An `EJBObject` stub tracks available replicas of an EJB in a cluster.

Stateless Session Beans

When a home creates a stateless bean, it returns a `EJBObject` stub that lists all of the servers in the cluster, to which the bean should be 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. 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.

Note: In WebLogic Server 8.1, new load balancing options provide server affinity for stateless EJB remote interfaces. To understand server affinity and how it affects load balancing and failover, see Round-Robin Affinity, Weight-Based Affinity, and Random-Affinity.

Stateful Session Beans

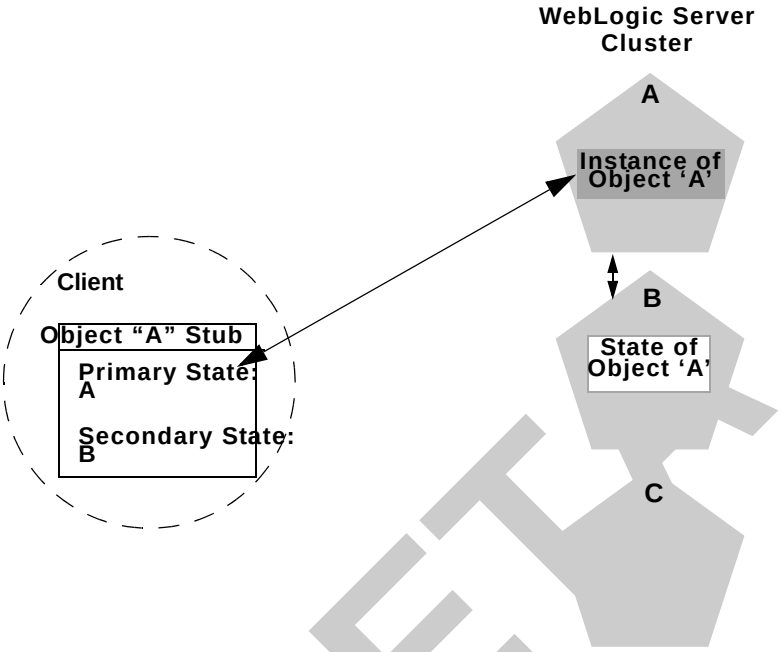
Method-level failover for a stateful service requires state replication. WebLogic Server satisfies this requirement by replicating the state of the primary bean instance to a secondary server instance, using a replication scheme similar to that used for HTTP session state.

When a home interface creates a stateless session bean instance, it selects a secondary instance to host the replicated state, using the same rules defined in "Using Replication Groups" on page 5-5. The home interface returns a `EJBObject` stub to the client that lists the location of the primary bean instance, and the location for the replicated bean state.

The following figure shows a client accessing a clustered stateful session EJB.

BETA

Figure 5-5 Client Accessing Stateful Session EJB



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.

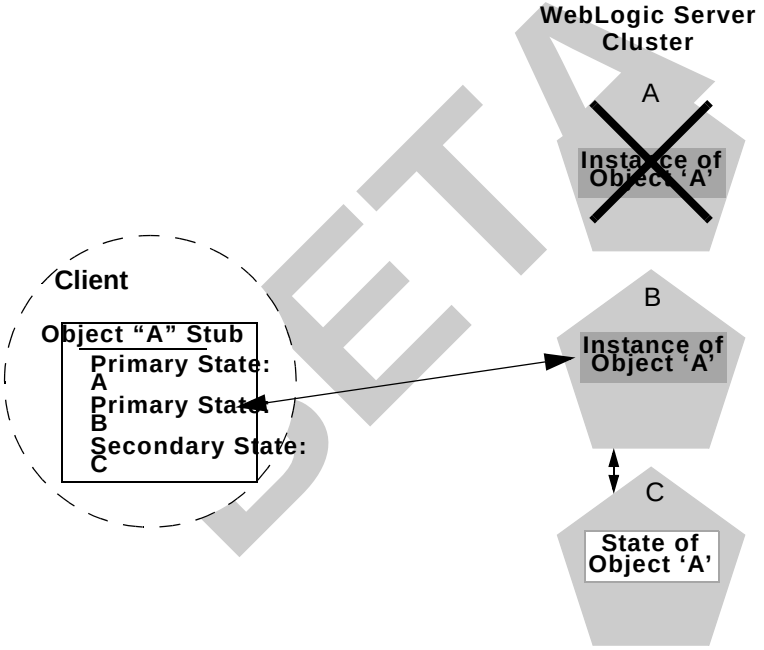
BETA

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.

Figure 5-6 Replica Aware Stubs are Updated after Failover



Entity EJBs

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

- Read-Write 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.

- Read-Only Entities

  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.

Failover for Entity Beans and EJB Handles

Failover for entity beans and EJB handles depends upon the existence of the cluster address. You can explicitly define the cluster address, or allow WebLogic Server to generate it automatically, as described in "Cluster Address" on page 7-6. If you explicitly define cluster address, you must specify it 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.

Clustering Support for 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 clusters, see "WebLogic RMI Features and Guidelines" in *Programming WebLogic RMI*.

Object Deployment Requirements

If you are programming EJBs to be used in a WebLogic Server cluster, read the instructions in this section 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 Descriptor Reference" in *Programming WebLogic Enterprise JavaBeans* 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 Environment" in *Programming WebLogic JNDI* to understand the implications of binding clustered objects in the JNDI tree.

BETA

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.

Server Migration

In a WebLogic Server cluster, most services are deployed homogeneously on all the server instances in the cluster, enabling transparent failover from one server to another. In contrast, "pinned services" such as JMS and the JTA transaction recovery system are targeted at individual server instances within a cluster—for these services, WebLogic Server supports failure recovery with *migration*, as opposed to failover.

In previous releases of WebLogic Server, JMS servers and the JTA transaction recovery system could be migrated manually upon failure of the hosting server instance. This feature is still supported, and is described in "Migration for Singleton Services" on page 5-31.

WebLogic Server 9.0 provides a new feature for making JMS and the JTA transaction system highly available: *migratable servers*. Migratable servers provide for both automatic and manual migration at the server-level, rather than the service level.

Note: Server-level migration is an *alternative* to service-level migration. Service migration and server migration are not intended to be used in combination—if you migrate an individual service within your cluster, do not also migrate an entire server instance.

A migratable server is a clustered server instance that migrates in its entirety, along with all the services it hosts. Migratable servers are intended to host pinned services, such as JMS servers and the JTA transaction recovery servers, but they can also host clusterable services. All services that run on a migratable server are highly available.

When a migratable server becomes unavailable for any reason, for instance, if it hangs, loses network connectivity, or its host machine fails—migration is automatic. Upon failure, a migratable server is automatically restarted on the same machine if possible. If the migratable server cannot be restarted on the machine where it failed, it is migrated to another machine. In addition, an administrator can manually initiate migration of a server instance.

Tips for Configuring Server Migration

This section contains tips for setting up server migration.

BETA

Before You Start WebLogic Server

- Give superuser privileges to the `wlifconfig.sh` file. (Located in the `$BEA_HOME/weblogic90b/common/bin directory`.) The script must be able to run ifconfig on your machines, which is usually a super-user only program.
- You may need to set `#!/bin/ksh` at the top of `wlscontrol.sh` and `wlifconfig.sh` (both located in the `$BEA_HOME/weblogic90b/common/bin` directory) to a different default shell, if you do not have ksh installed. Any POSIX-compliant shell will work.
- Copy `wlifconfig.sh`, `wlscontrol.sh` and `nodemanager.domains` to a directory that is part of your machines' PATH. The `.sh` files are located in `$BEA_HOME/weblogic90b/common/bin`. `nodemanager.domains` is located in `$BEA_HOME/weblogic90b/common/nodemanager`.
- The machines that will host a migratable server must all trust each other. It should be possible to type '`ssh/rsh machineA`' from machineB and get to a shell prompt without having to enter username/password, and vice versa.
- Set `Interface` in `wlscontrol.sh` to the name of the network interfaces on your machines. (Usually looks something like '`hme0'. Running 'ifconfig -a`' and will list all of the addresses and interfaces on your machine.)
- You need one IP address for each server that will be migrated. This IP address should not be tied to any machine. (When migration occurs, the IP will be removed from the machine in question and enabled on the new host machine for the server.)
- You must have a reliable database. The server instances will only be as reliable as the database is. For experimental purposes, a normal database will suffice. For a production environment, only HA databases are recommended. (If the database goes down, all the servers will shut themselves down.)

WebLogic Server Configuration (after you start the Administration Server)

- Each server that will be migrated should have one of the floating IPs set as its Listen Address. Any server that has one of these floating IPs must also have `AutoMigrationEnabled` set to true. (Servers with this attribute set to true will be referred to as Migratable Servers in this document.)
- You should set up a JDBC Data Source that points to the previously mentioned database.
- Set the Data Source For Automatic Migration attribute on the cluster to the name of your Data Source.

BETA

- Configure a Machine MBean (under Machines, in the Administration Console) for each machine that the Migratable Servers could reside on. Make sure each machine has the ssh/rsh nodemanager configured correctly.
- Set the Candidate Machines for the Servers that will be migrating. Each server can have a different set of Candidate machines, or they can all have the same set.
- Currently, you cannot create Channels/NetworkAccessPoints that have a different Listen Address on a migratable server.
- There is no built-in mechanism for transferring files that a server depends on between machines. Using a disk accessible from all machines is the preferred way to ensure file availability.

Use High Availability Storage for State Data

The server migration process migrates services, but not the state information associated with work in process at the time of failure.

To ensure high availability, it is critical that such state information remains available to the server instance and the services it hosts after migration. Otherwise, data about the work in process at the time of failure may be lost. State information maintained by a migratable server, such as the data contained in transaction logs, should be stored in a shared storage system that is accessible to any potential machine to which a failed migratable server might be migrated. For highest reliability, use a shared storage solution that is itself highly available—for example, a storage area network (SAN).

In addition, the *lease table,* described in the following sections, which is used to track the health and liveness of migratable servers should also stored in a high availability database.

BETA

Server Migration Processes and Communications

The server migration process involves the following WebLogic Server services and resources:

- Migratable Servers—You configure one or more clustered Managed Servers that host pinned services as migratable servers.
- Cluster Master—One server instance in a cluster that contains migratable servers acts a the *cluster master* and orchestrates the process of automatic server migration, in the event of failure. Any Managed Server in a cluster can server as the cluster master, whether it hosts pinned services or not.

- Target Machines—A set of machines that are designated as allowable or preferred hosts for migratable servers.
- Node Manager—Node Manager is used by the Administration Server or a stand-alone Node Manager client, to start and stop migratable servers, and is invoked by the cluster master to shutdown and restart migratable servers, as necessary.

  For background information about Node Manager and how it fits into a WebLogic Server environment, see "Using Node Manager to Control WebLogic Server" in *Configuring WebLogic Server Environments.*
- Lease table—You configure a database table in which migratable servers persist their state, and which the cluster master monitors to verify the health and liveness migratable servers.
- Administration Server—Used to configure migratable servers and target machines, to obtain the runtime state of migratable servers, and to orchestrate the manual migration process.

The sections that follow describe key processes in a cluster that contains migratable servers:

- "Startup Process in a Cluster with Migratable Servers" on page 5-22
- "Automatic Migration Process" on page 5-24
- "Manual Migration Process" on page 5-26

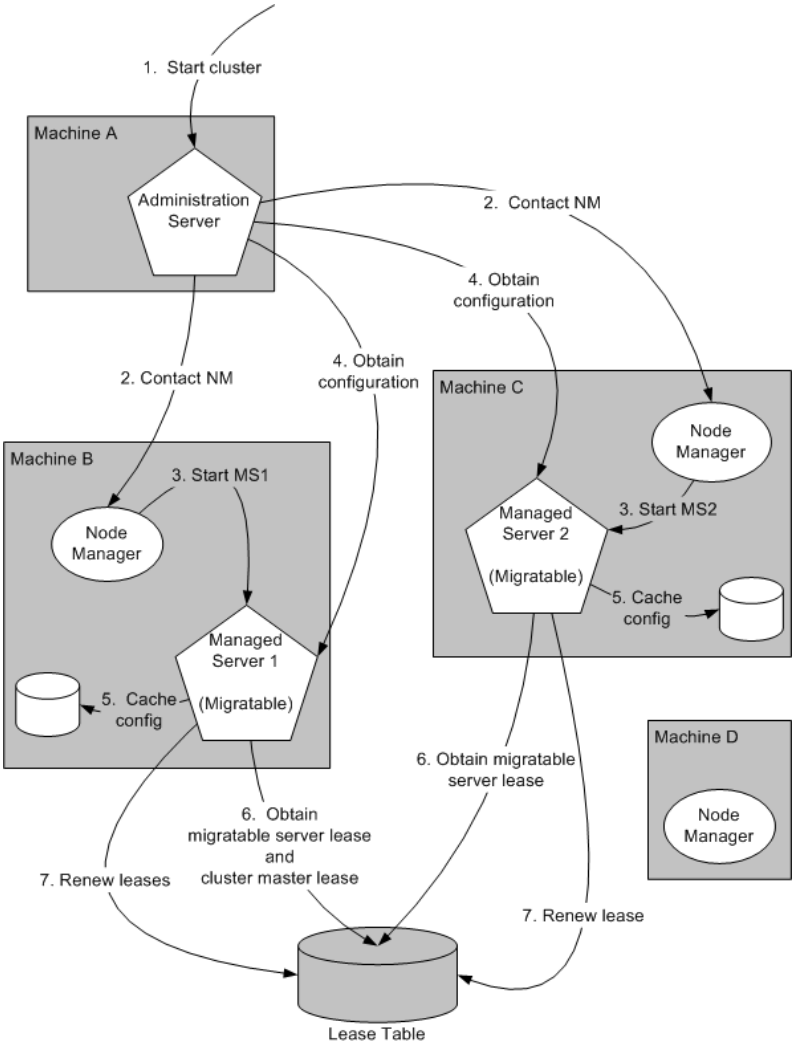
Startup Process in a Cluster with Migratable Servers

Figure 5-7, "Startup of Cluster With Migratable Servers," on page 5-23 illustrates the processing and communications that occur during startup of a cluster that contains migratable servers.

The example cluster contains two Managed Servers, both of which are migratable. The Administration Server and the two Managed Servers each run on different machines. A fourth machine is available as a backup—in the event that one of the migratable servers fails. Node Manager running on the backup machine and on each machine with a running migratable server.

BETA

Figure 5-7 Startup of Cluster With Migratable Servers



These are the key steps that occur during startup of the cluster illustrated in Figure 5-7:

1. The administrator starts up the cluster.
2. The Administration Server invokes Node Manager on Machines B and C to start Managed Servers 1 and 2, respectively. For more information, see "Administration Server's Role in Server Migration" on page 5-28.

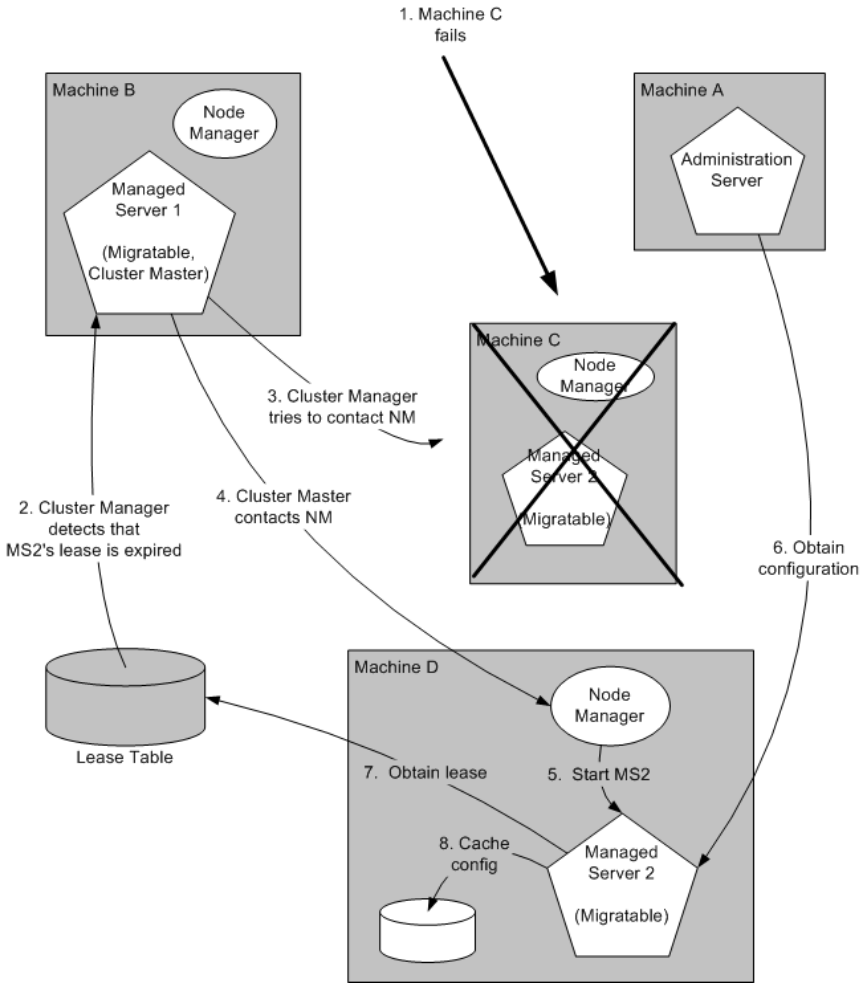
3. The Node Manager on each machine starts up the Managed Server that runs there. For more information, see “Node Manager’s Role in Server Migration” on page 5-29.
4. Managed Servers 1 and 2 each start up, and contact the Administration Server for their configuration. For more information, see “Migratable Server Behavior in a Cluster” on page 5-28.
5. Managed Servers 1 and 2 cache the configuration they started up.
6. Managed Servers 1 and 2 each obtain a migratable server lease in the lease table. Because Managed Server 1 starts up first, it also obtains a cluster master lease. For more information, see “Cluster Master’s Role in Server Migration” on page 5-30.
7. Managed Server 1 and 2 periodically renew their leases in the lease table, proving their health and liveness.

Automatic Migration Process

Figure 5-8, “Automatic Migration of a Failed Server,” on page 5-25 illustrates the automatic migration process after failure of the machine hosting Managed Server 2.

BETA

Figure 5-8 Automatic Migration of a Failed Server



1. Machine C, which hosts Managed Server 2, fails.
2. Upon its next periodic review of the lease table, the cluster master detects that Managed Server 2's lease has expired. For more information, see "Cluster Master's Role in Server Migration" on page 5-30.
3. The cluster master tries to contact Node Manager on Machine C to restart Managed Server 2, but fails, because Machine C is unreachable.

Note: If the Managed Server 2's lease had expired because it was hung, and Machine C was reachable, the cluster master would use Node Manager to restart Managed Server 2 on Machine C.

4. The cluster master contacts Node Manager on Machine D, which is configured as an available host for migratable servers in the cluster.
5. Node Manager on Machine D starts Managed Server 2. For more information, see "Node Manager's Role in Server Migration" on page 5-29.
6. Managed Server 2 starts up and contacts the Administration Server to obtain its configuration.
7. Managed Server 2 caches the configuration it started up with.
8. Managed Server 2 obtains a migratable server lease.

During migration, the clients of the Managed Server that is migrating may experience a brief interruption in service; it may be necessary to reconnect. On Solaris and Linux operating systems, this can be done using `ifconfig` command. The clients of a migrated server do not need to know the particular machine to which it has migrated.

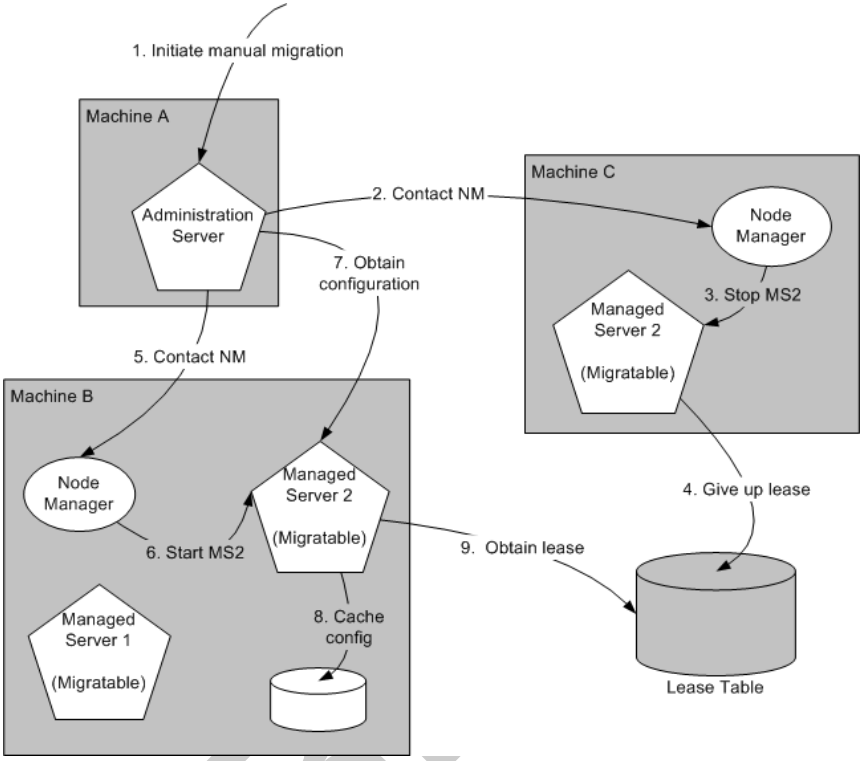
When a machine from a server was migrated becomes available again, reversal of the migration process—migrating the server instance back to its original host machine—is referred to as *failback*. WebLogic Server does not automate the process of failback. An administrator can accomplish failback by manually the server instance back to its original host.

Manual Migration Process

Figure 5-9, "Manual Server Migration," on page 5-27 illustrates what happens when an administrator manually migrates a migratable server.

BETA

Figure 5-9 Manual Server Migration



1. An administrator uses the Administration Console to initiate the migration of Managed Server 2 from Machine C to Machine B.
2. The Administration Server contacts Node Manager on Machine C. For more information, see "Administration Server's Role in Server Migration" on page 5-28.
3. Node Manager on Machine C stops Managed Server 2.
4. Managed Server 2 removes its row from the lease table.
5. The Administration Server invokes Node Manager on Machine B.
6. Node Manager on Machine B starts Managed Server 2.
7. Managed Server 2 obtains its configuration from the Administration Server.
8. Managed Server 2 caches the configuration it started up with.

9. Managed Server 2 adds a row to the lease table.

Administration Server's Role in Server Migration

In a cluster that contains migratable servers, the Administration Server:

- Invokes Node Manager, on each machine that hosts cluster members, to start up the migratable serves. This is a prerequisite for server migratability—if a server instance was not initially started by Node Manager, it cannot be migrated.
- Invokes Node Manager on each machine involved in a manual migration process to stop and start the migratable server.
- Invokes Node Manager on each machine that hosts cluster members to stop server instances during a normal shutdown. This is a prerequisite for server migratability—if a server instance is shutdown directly, without using Node Manager, when the cluster master detects that the server instance is not running, it will call Node Manager to restart it.

In addition, the Administration Server provides its regular domain management functionality, persisting configuration updates issued by an administrator, and providing a run-time view of the domain, including the migratable servers it contains.

Migratable Server Behavior in a Cluster

A migratable server is a clustered Managed Server that has been configured as migratable. These are the key behaviors of a migratable server:

- Upon startup and restart by Node Manager, a migratable server adds a row to the lease table. The row for a migratable server contains a timestamp, and the machine where it is running.
- When a migratable server adds a row to the database as a result of startup, it tries to take on the role of cluster master, and succeeds if it is the first server instance to join the cluster.
- Periodically, the server renews its "lease" by updating the timestamp in the lease table.

  By default a migratable server renews its lease every 30,000 milliseconds—the product of two configurable `ServerMBean` properties:

  - `HealthCheckIntervalMillis`, which by default is 10,000.
  - `HealthCheckPeriodsUntilFencing`, which by default is 3.

- If a migratable server fails to reach the lease table and renew its lease before the lease expires, it terminates as quickly as possible using a Java `System.exit`—in this case, the

BETA

lease table still contains a row for that server instance. For information about how this relates to automatic migration, see "Cluster Master's Role in Server Migration" on page 5-30.

- Upon a suspend, a shutdown, or a force shutdown by its local Node Manager process, a migratable server removes the row that it created at startup in the lease table.
- During operation, a migratable server listens for heartbeats from the cluster master. When it detects that cluster master is not sending heartbeats, it attempts to take over the role of cluster master, and succeeds if no other server instance has claimed that role.

Node Manager's Role in Server Migration

The use of Node Manager is required for server migration—it must run on each machine that hosts, or is intended to host.

Node Manager supports server migration in these ways:

- Node Manager must be used for initial startup of migratable servers.

  When you initiate the startup of a Managed Server from the Administration Console, the Administration Server uses Node Manager to start up the server instance. You can also invoke Node Manager to start the server instance using the stand-alone Node Manager client, however, the Administration Server must be available so that the Managed Server can obtain its configuration.

  Note: Migration of a server instance that not initially started with Node Manager will fail.

- Node Manager must be used for suspend, shutdown, or force shutdown of migratable servers.
- Node Manager tries to restart a migratable server whose lease has expired on the machine where it was running at the time of failure.

  Node Manager performs the steps in the server migrate process by running customizable shell scripts, provided with WebLogic Server, that start, restart and stop servers; migrate IP addresses; and mount and unmount disks. The scripts are available for Solaris, Linux and Windows.

  – In an automatic migration, the cluster master invokes Node Manager to perform the migration.

  – In a manual migration, the Administration Server invokes Node Manager to perform the migration.

BETA

Cluster Master's Role in Server Migration

In a cluster that contains migratable servers, one server instance acts as the cluster master—whose role is to orchestrate the server migration process. Any server instance in the cluster can serve as the cluster master. When you start a cluster that contains migratable servers, the first server to join the cluster becomes the cluster master and starts up the cluster manager service. If a cluster does not include at least one migratable server, it does not require a cluster manager, and the cluster master service does not start up. In the absence of a cluster master, migratable servers can continue to operate, but server migration is not possible. These are the key functions of the cluster master:

- Issue periodic heartbeats to the other servers in the cluster.
- Periodically read the lease table to verify that each migratable server has a current lease. An expired lease indicates to the Cluster Master that the migratable server should be restarted.
- Upon determining that a migratable server's lease is expired, wait for period specified by the `FencingGracePeriodMillis` on the `ClusterMBean`, and then try to invoke the Node Manager process on the machine that hosts the migratable server whose lease is expired, to restart the migratable server.
- If unable to restart a migratable server whose lease has expired on its current machine, the cluster master selects a target machine in this fashion:
  - If you have configured a list of preferred destination machines for the migratable server, the cluster master chooses a machine on that list, in the order the machines are listed.
  - Otherwise, the cluster master chooses a machine on the list of those configured as available for hosting migratable servers in the cluster.

  A list of machines that can host migratable servers can be configured at two levels: for the cluster as a whole, and for an individual migratable server. You can define a machine list at both levels. You must define a machine list at least one level.
- To accomplish the migration of a server instance to a new machine, the cluster master invokes the Node Manager process on the target machine to create a process for server instance,.

  The time required to perform the migration depends on the server configuration and startup time.

BETA

– The maximum time taken for cluster master to restart the migratable server is (`HealthCheckPeriodsUntilFencing` * `HealthCheckIntervalMillis`) + `FencingGracePeriodMillis`.

– The total time before the server becomes available for client requests depends on the server startup time and the application deployment time.

Migration for Singleton Services

In a WebLogic Server cluster, most services are deployed homogeneously on all the server instances in the cluster, enabling transparent failover from one server to another. In contrast, *singleton* services, such as JMS and the JTA transaction recovery system, run only on one server in the cluster at any given time.

WebLogic Server allows the administrator to migrate singleton services from one server to another in the cluster, either in response to a server failure or as part of regularly-scheduled maintenance. This capability improves the availability of singleton services in a cluster, because those services can be quickly restarted on a redundant server should the host server fail.

WebLogic Server supports service-level migration for JMS servers and the JTA transaction recovery service. This document refers to these services as *migratable services*, because you can move them from one server to another within a cluster. Note that JMS also offers improved service continuity in the event of a single Weblogic Server failure by enabling you to configure multiple physical destinations (queues and topics) as part of a single distributed destination set.

Note: WebLogic Server 9.0 also supports migration at the server level—a complete server instance, and all of the services it hosts can be migrated to another machine, either automatically, or manually. This feature is described in "Server Migration" on page 5-19.

How Migration of Pinned Services Works

Clients access a migratable service in a cluster using a migration-aware RMI stub. The RMI stub keeps track of which server currently hosts the pinned service, and it directs client requests accordingly. For example, when a client first accesses a pinned service, the stub directs the client request to the server instance in the cluster that currently hosts the service. If the service migrates to a different WebLogic Server between subsequent client requests, the stub transparently redirects the request to the correct target server.

WebLogic Server implements a migration-aware RMI stub for JMS servers and the JTA transaction recovery service when those services reside in a cluster and are configured for migration.

BETA

Migrating a Service When Currently Active Host is Unavailable

There are special considerations when you migrate a service from a server instance that has crashed or is unavailable to the Administration Server. If the Administration Server cannot reach the previously active host of the service at the time you perform the migration, that Managed Server's local configuration information will not be updated to reflect that it is no longer the active host for the service. In this situation, you must purge the unreachable Managed Server's local configuration cache before starting it again. This prevents the previous active host from re-activating at startup a service that has been migrated to another Managed Server. For more information see "Migrating When the Currently Active Host is Unavailable" on page 7-28.

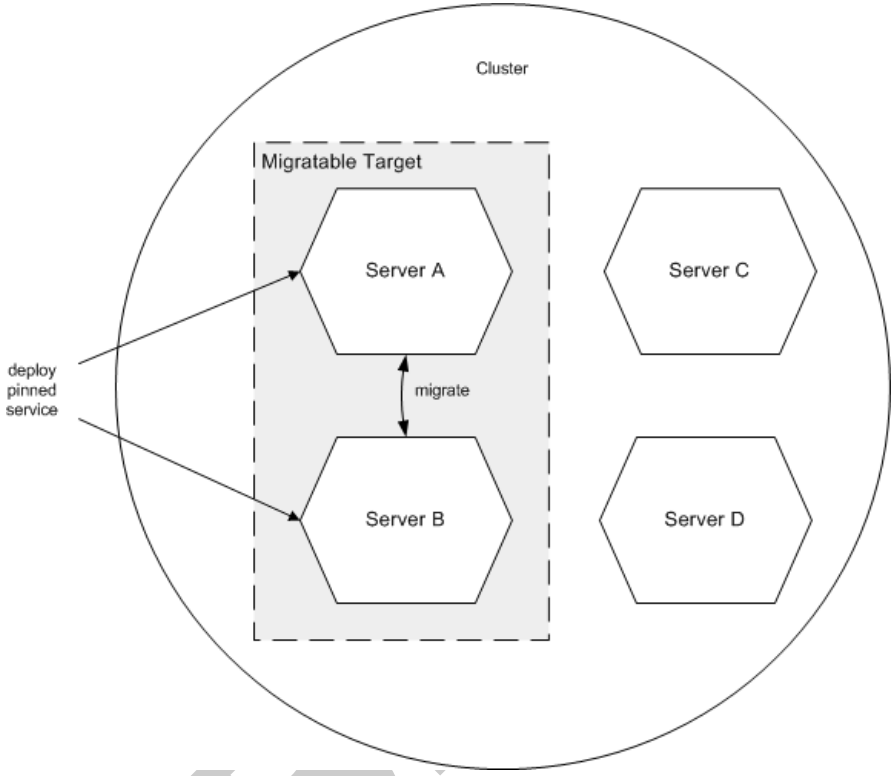
Defining Migratable Target Servers in a Cluster

By default, WebLogic Server can migrate the JTA transaction recovery service or a JMS server to any other server in the cluster. You can optionally configure a list of servers in the cluster that can potentially host a pinned service. This list of servers is referred to as a *migratable target*, and it controls the servers to which you can migrate a service. In the case of JMS, the migratable target also defines the list of servers to which you can deploy a JMS server.

For example, the following figure shows a cluster of four servers. Servers A and B are configured as the migratable target for a JMS server in the cluster.

BETA

Figure 5-10 Migratable Target in Cluster



In the above example, the migratable target allows the administrator to migrate the pinned JMS server only from Server A to Server B, or vice versa. Similarly, when deploying the JMS server to the cluster, the administrator selects either Server A or B as the deployment target to enable migration for the service. (If the administrator does not use a migratable target, the JMS server can be deployed or migrated to any available server in the cluster.)

WebLogic Server enables you to create separate migratable targets for the JTA transaction recovery service and JMS servers. This allows you to always keep each service running on a different server in the cluster, if necessary. Conversely, you can configure the same selection of servers as the migratable target for both JTA and JMS, to ensure that the services remain co-located on the same server in the cluster.

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.

For instructions on clustering JDBC objects, see "Configure Clustered JDBC" on page 7-22.

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.

BETA

CHAPTER 6

Cluster Architectures

BETA

This following sections describe alternative architectures for a WebLogic Server cluster:

Architectural and Cluster Terminology

This section defines terms used in this document.

Architecture

In this context the *architecture* refers to how the tiers of an application are deployed to one or more clusters.

Web Application Tiers

A Web application is divided into several "tiers" that correspond to the logical services the application provides. Because not all Web applications are alike, 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.

Combined Tier Architecture

A cluster architecture in which all tiers of the Web application are deployed to a single WebLogic Server cluster is called a combined tier architecture.

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.

BETA

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 such as those from Alteon WebSystems, which may 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 "For a discussion of external load balancers, session cookie persistence, and the WebLogic Server session cookie, see "Load Balancing HTTP Sessions with an External Load Balancer" on page 4-2" on page 7-13 for more information.

Proxy Plug-In

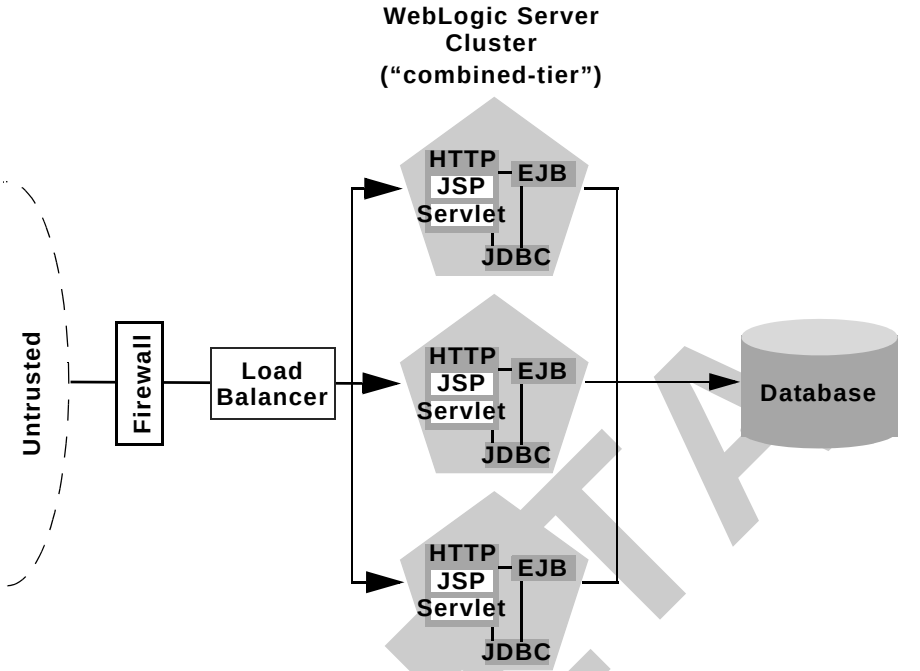
A *proxy plug-in* is a WebLogic Server extension to an HTTP server—such as Apache, Netscape Enterprise Server, or Microsoft Internet Information Server—that accesses clustered servlets provided by a WebLogic Server cluster. The proxy plug-in contains the load balancing logic for accessing servlets and 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 the session state fails.

Recommended Basic Architecture

The recommended basic architecture is a combined tier architecture—all tiers of the Web application are deployed to the same WebLogic Server cluster. This architecture is illustrated in the following figure.

BETA

Figure 6-1 Recommended Basic Architecture



The benefits of the Recommended Basic Architecture are:

- Ease of 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 WebLogic Server 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 can 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.

BETA

- 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). For more information about load balancers, see "For a discussion of external load balancers, session cookie persistence, and the WebLogic Server session cookie, see "Load Balancing HTTP Sessions with an External Load Balancer" on page 4-2" on page 7-13.

When Not to Use a Combined Tier Architecture

While a combined tier architecture, such as the Recommended Basic Architecture, meets the needs of many Web applications, it limits your ability to fully employ the load balancing and failover capabilities of a cluster. Load balancing and failover can be introduced only at the interfaces between Web application tiers, so, when tiers are deployed to a single cluster, you can only load balance between clients and the cluster.

Because most load balancing and failover occurs between clients and the cluster itself, a combined tier architecture meets the 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, as described in the following section.

Consider the frequency of invocations of the object tier by the presentation tier when deciding between a combined tier and multi-tier architecture. If presentation objects usually invoke the object tier, a combined tier architecture may offer better performance than a multi-tier architecture.

BETA

Recommended Multi-Tier Architecture

This section describes the Recommended Multi-Tier Architecture, in which different tiers of your application are deployed to different clusters.

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:

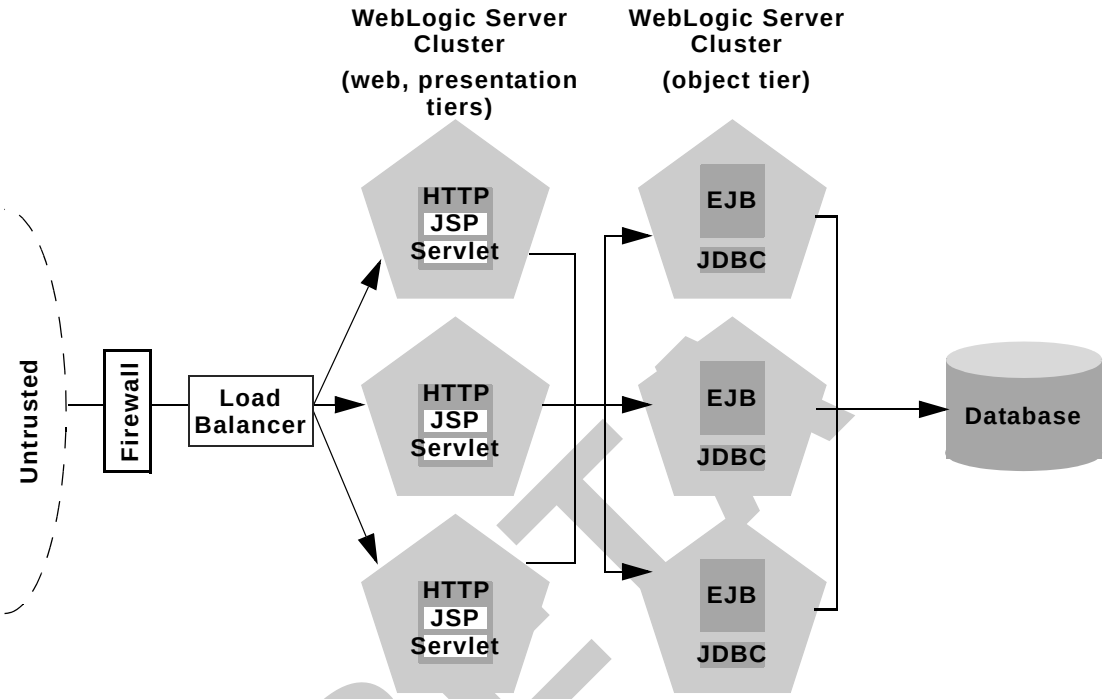
- Require load balancing for method calls to clustered EJBs.
- Require more flexibility for balancing the load between servers that provide HTTP content and servers that provide clustered objects.
- Require higher availability (fewer single points of failure).

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.

BETA

Figure 6-2 Recommended Multi-Tier Architecture



Physical Hardware and Software Layers

In the Recommended Multi-Tier Architecture the application tiers are hosted on two separate physical layers of hardware and software.

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 "Optimization for Collocated Objects" on page 4-12. However, you gain the ability to load balance on each method call to certain clustered objects, as described in the following section.

Benefits of Multi-Tier Architecture

The multi-tier architecture provides these advantages:

- 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 Clustered Objects in a in Multi-Tier Architecture" on page 6-8.

- 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. For more information, see "Security Options for Cluster Architectures" on page 6-16.

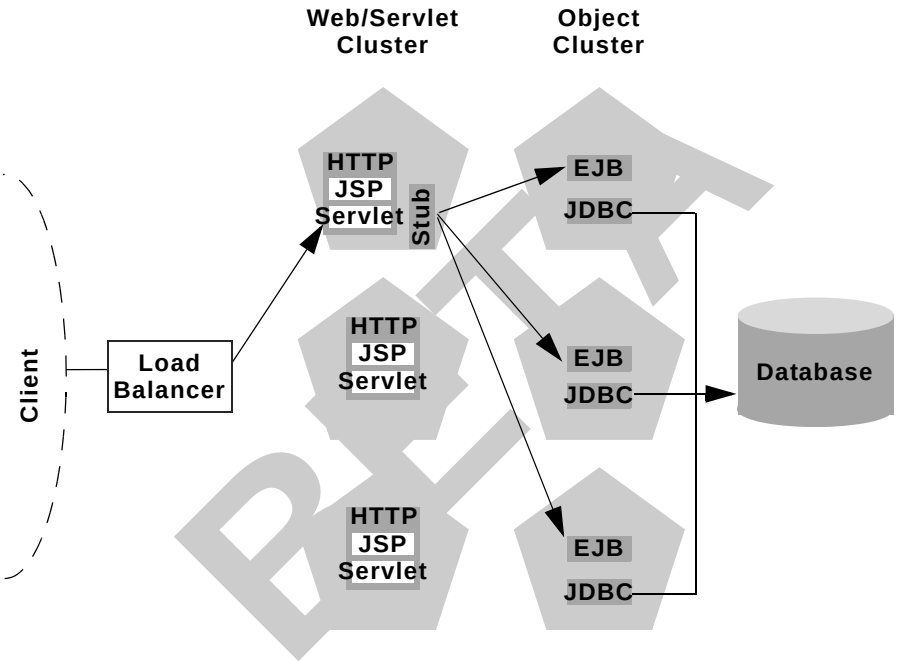
Load Balancing Clustered Objects in a in Multi-Tier Architecture

WebLogic Server's collocation optimization for clustered objects, described in "Optimization for Collocated Objects" on page 4-12, 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.

BETA

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 (described in "Optimization for Collocated Objects" on page 4-12), 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.

Figure 6-3 Load Balancing Objects in a 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 Descriptor Reference" in *Programming WebLogic Enterprise JavaBeans* 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 Architecture), 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 Considerations for Multi-Tier Architecture

IP Socket Usage

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 Figure 6-2, 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. For details, see "Configuring Reader Threads for Java Socket Implementation" on page 2-5.

BETA

Hardware Load Balancers

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. For details, see "Configure Load Balancing Method for EJBs and RMIs" on page 7-12.

Limitations of Multi-Tier Architectures

This section summarizes the limitations of multi-tier cluster architectures.

No Collocation Optimization

Because the Recommended Multi-Tier Architecture 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" on page 6-8.

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.

Use of `ExternalDNSName` is required for configurations in which a firewall is performing Network Address Translation, unless clients are accessing WebLogic Server using t3 and the default channel. For instance, `ExternalDNSName` is required for configurations in which a firewall is performing Network Address Translation, and clients are accessing WebLogic Server using HTTP via a proxy plug-in.

BETA

Recommended Proxy Architectures

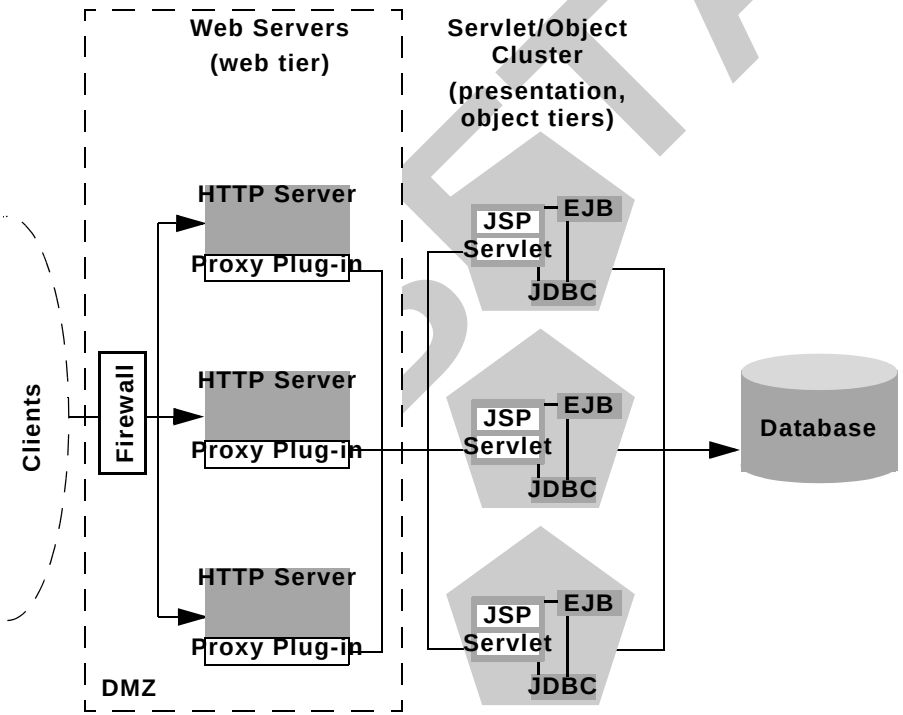
You can configure WebLogic Server clusters to operate alongside existing Web servers. In such an architecture, a bank of Web servers provides static HTTP content for the Web application, using a WebLogic proxy plug-in or `HttpClusterServlet` to direct servlet and JSP requests to a cluster.

The following sections describe two alternative proxy architectures.

Two-Tier Proxy Architecture

The two-tier proxy architecture illustrated in the following figure is similar to the "Recommended Basic Architecture" on page 6-3, except that static HTTP servers are hosted on a bank of Web servers.

Figure 6-4 Two-Tier Proxy Architecture



BETA

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 single machine or on multiple separate machines.

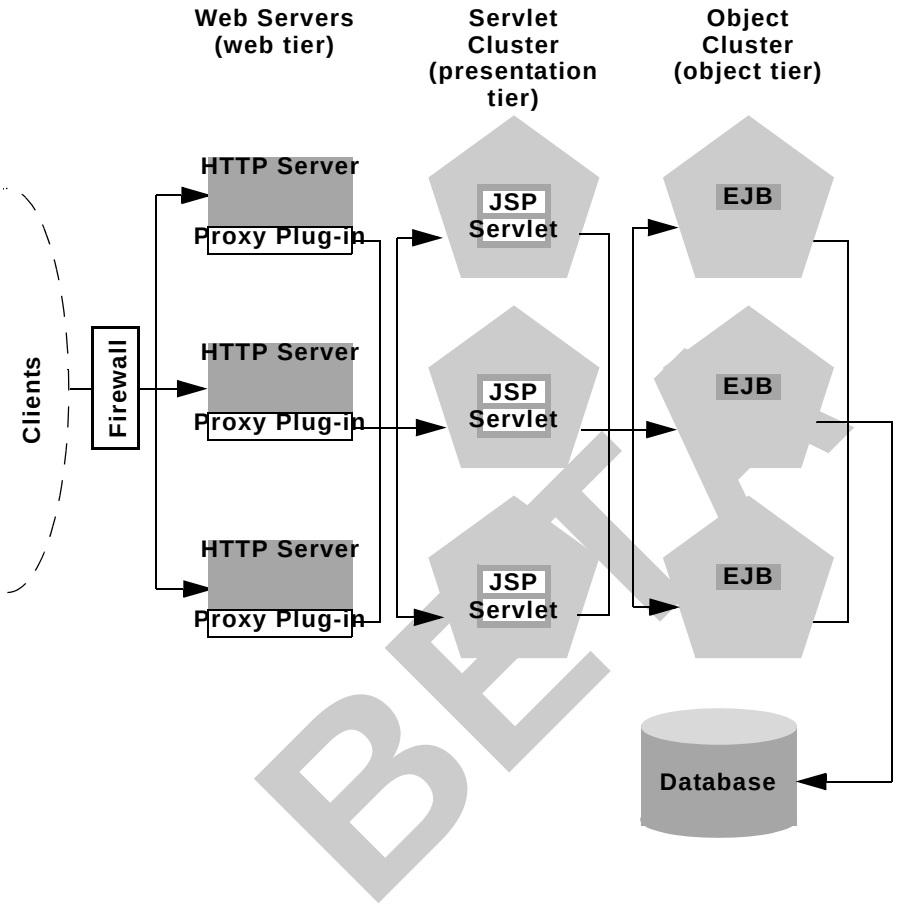
The Servlet/Object layer differs from the combined-tier cluster described in Recommended Basic Architecture in that it does not provide static HTTP content to application clients.

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 following figure.

BETA

Figure 6-5 Multi-Tier Proxy Architecture



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 Benefits

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.

Proxy Architecture Limitations

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.

BETA

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.

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.

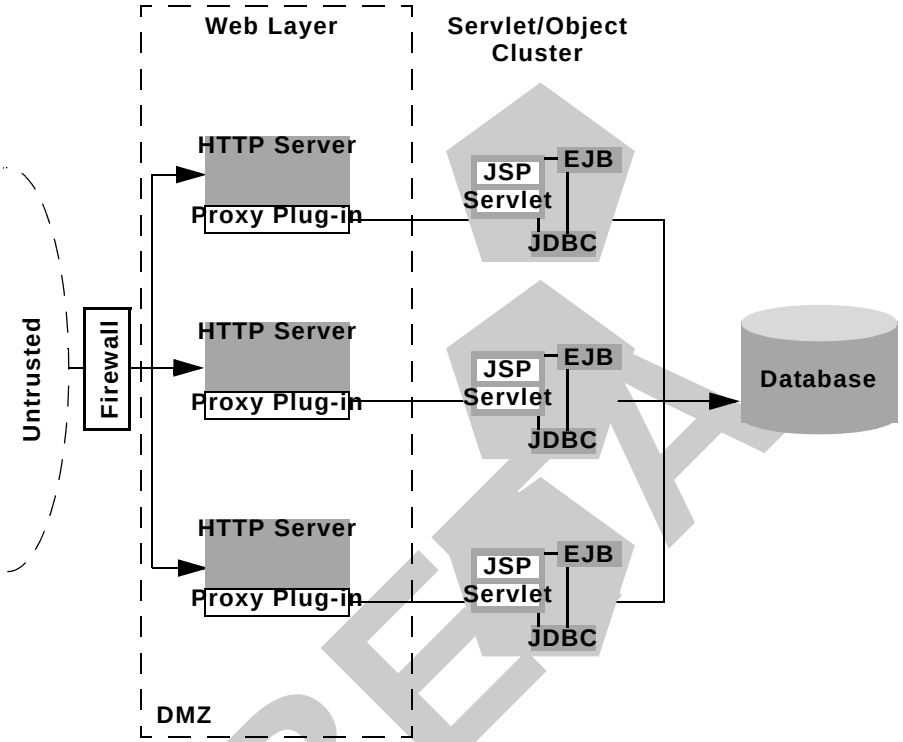
The sections that follow describe several common ways of defining your DMZ to create varying levels of application security.

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 Recommended Basic Architecture or Recommended Multi-Tier Architecture cluster architectures.

BETA

Figure 6-6 Basic Proxy with Firewall Architecture



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.

Firewall Between Proxy Layer and Cluster

If you place a firewall between the proxy layer and the cluster, follow these configuration guidelines:

- Bind to clustered server instances using publicly-listed DNS names, rather than IP addresses, to ensure that the proxy plug-ins can connect to each server in the cluster without address translation error that might otherwise occur, as described in "Firewall Considerations" on page 8-13.
- If the internal and external DNS names of a clustered server instance are not identical, use the `ExternalDNSName` attribute for the server instance to define the its 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*.

Note: If the clustered servers segregate https and http traffic on a pair of custom channels, see "Channels, Clusters, and Firewalls" in *Designing and Configuring WebLogic Server Environments.*

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.

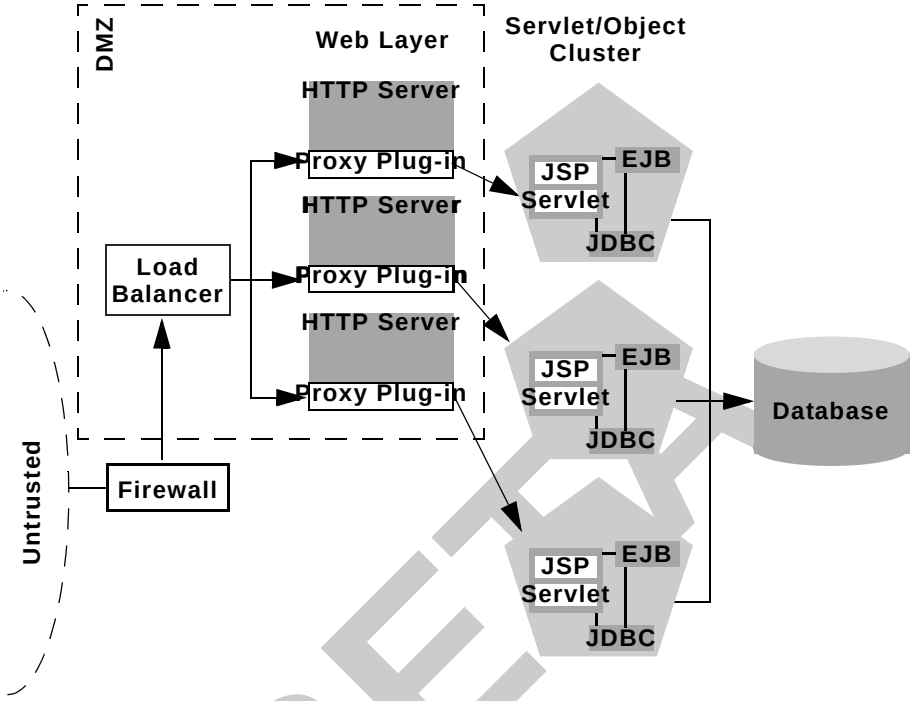
If you choose to define your DMZ more conservatively, you can place additional firewalls using the information in "Additional Security for Shared Databases" on page 6-21.

Combining Firewall with Load Balancer

If you use load balancing hardware with a recommended cluster architecture, 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.

BETA

Figure 6-7 Basic Proxy with Firewall and Load Balancer Architecture



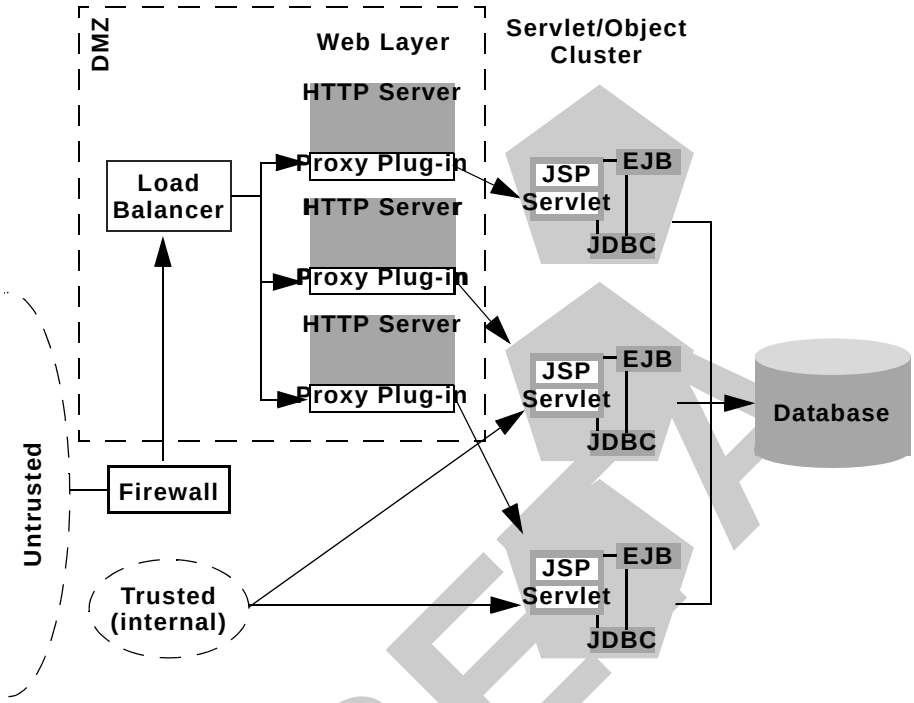
The above setup places the load balancer within the DMZ along with the web tier. Using a firewall in this configuration can simplify security policy administration, because the firewall need only limit access to the load balancer. This setup can also simplify administration for sites that support internal clients to the Web application, as described below.

Expanding the Firewall for Internal Clients

If you support internal clients that require direct access to 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.

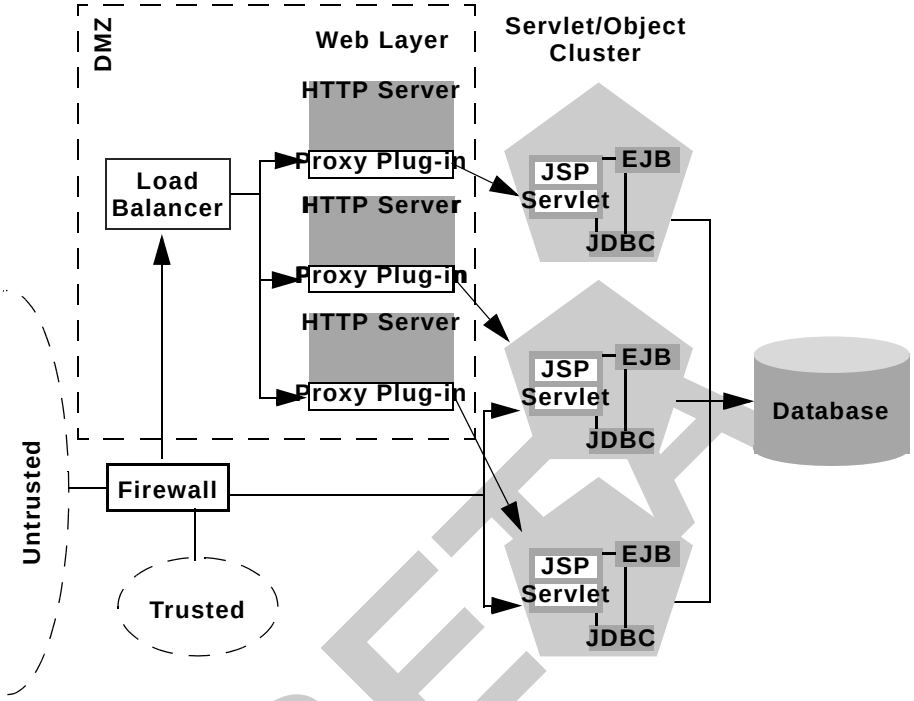
Figure 6-8 VPN Users have Restricted Access Through Firewall



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 Server instances hosting the presentation tier, as shown in the following figure.

BEA

Figure 6-9 Application Components Have Restricted Access Through Firewall



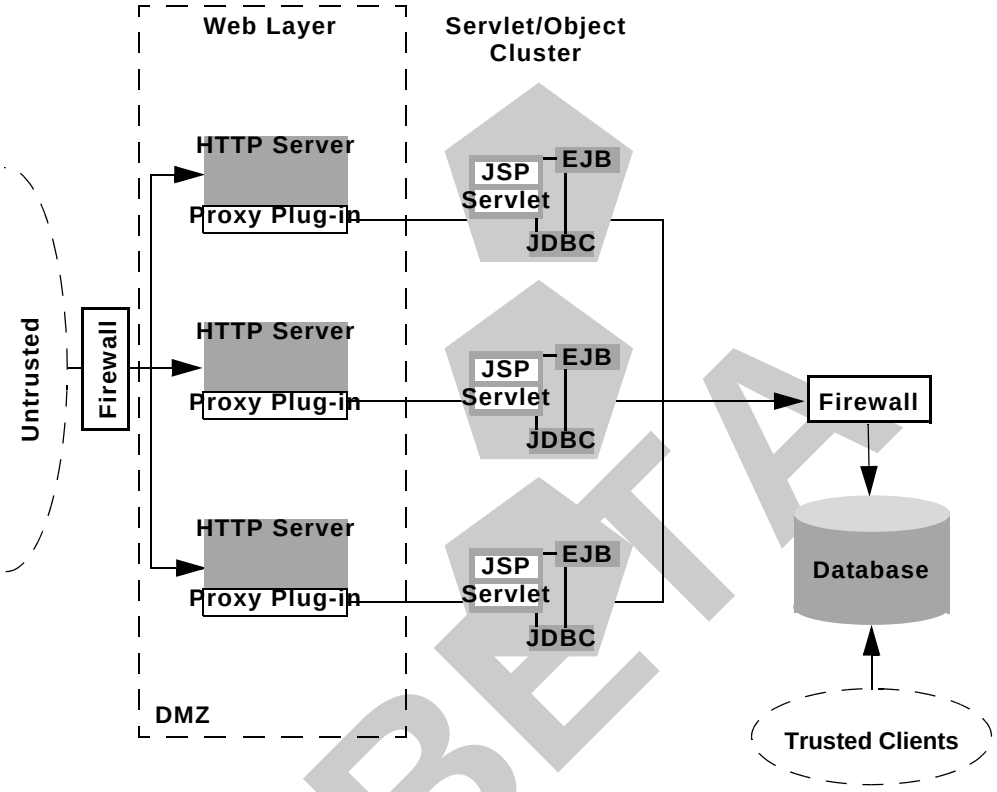
Additional Security for Shared Databases

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" on page 6-16 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.

Figure 6-10 DMZ with Two Firewalls Architecture



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.

BETA

CHAPTER 7

Setting up WebLogic Clusters

BETA

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

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.

For information about the configuration facilities available in WebLogic Server and the tasks they support, see "Understanding Cluster Configuration and Application Deployment" on page 3-1.

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 "Cluster Architectures" on page 6-1, and "Load Balancing in a Cluster" on page 4-1.

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 6-16.

Notes: Some network topologies can interfere with multicast communication. If you are deploying a cluster across a WAN, see "If Your Cluster Spans Multiple Subnets in a WAN" on page 2-2.

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 2-3.

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.

BETA

System and software prerequisites are listed in "Preparing to Install WebLogic Platform" in *Installing BEA WebLogic Platform*.

WebLogic Server allows you to set up a cluster on a single, non-multihomed machine. This new capability is useful for demonstration or development environments.

> 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, 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" in *BEA WebLogic Server Performance and Tuning* for additional information.

Check Host Machines' Socket Reader Implementation

For best socket performance, configure the WebLogic Server host machine 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 2-4.

Setting Up a Cluster on a Disconnected Windows Machine

If you want to demonstrate a WebLogic Server cluster on a single, disconnected Windows machine, you must force Windows to load the TCP/IP stack. By default, Windows does not load the TCP/IP stack if it does not detect a physical network connection.

To force Windows to load the TCP/IP stack, disable the Windows media sensing feature using the instructions in "How to Disable Media Sense for TCP/IP in Windows" at http://support.microsoft.com/default.aspx?scid=kb;en-us;239924.

BETA

Identify Names and Addresses

During the cluster configuration process, you supply addressing information—IP addresses or DNS names, and port numbers—for the server instances in the cluster.

For information on intra-cluster communication, and how it enables load balancing and failover, see "WebLogic Server Communication in a Cluster" on page 2-1.

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 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 Considerations" on page 8-13.

BETA

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. Set this attribute in the Administration Console using the Environments—>Servers—>*ServerName*—>Configuration—>General tab, under the Advanced section.

Note: If clients are accessing WebLogic Server over the default channel and T3, do not set the `ExternalDNSName` attribute, even if the internal and external DNS names of a WebLogic Server instance are not identical.

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 Managed Servers in its domain. When you start a Managed Server, 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 combination of address and Listen Port number. Clustered server instances on a single non-multihomed machine can have the same address, but must use a different Listen Port.

BETA

Cluster Multicast Address and Port

Identify the address and port you will dedicate to multicast communications for your cluster. A multicast address is an IP address between 224.0.0.0 and 239.255.255.255.

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 6, "Cluster Architectures," 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.

BETA

Cluster Address

In WebLogic Server cluster, the *cluster address* is used in entity and stateless beans to construct the host name portion of request URLs.

You can explicitly define the cluster address when you configure the a cluster; otherwise, WebLogic Server dynamically generates the cluster address for each new request. Allowing WebLogic Server to dynamically generate the cluster address is simplest, in terms of system administration, and is suitable for both development and production environments.

Dynamic Cluster Address

If you do not explicitly define a cluster address when you configure a cluster, when a clustered server instance receives a remote request, WebLogic Server generates the cluster address, in the form:

```
listenaddress1:listenport1,listenaddress2:listenport2;listenaddress3:
listenport3
```

Each `listen address:listen port` combination in the cluster address corresponds to Managed Server and network channel that received the request.

- If the request was received on the Managed Server's default channel, the `listen address:listen port` combinations in the cluster address reflect the `ListenAddress` and `ListenPort` values from the associated `ServerMBean` and `SSLMBean` instances. For more information, see "The Default Network Channel" in *Configuring WebLogic Server Environments*.
- If the request was received on a custom network channel, the `listen address:listen port` in the cluster address reflect the `ListenAddress` and `ListenPort` values from `NetworkAccessPointMBean` that defines the channel. For more information about network channels in a cluster, see "Configuring Network Channels For a Cluster" in *Configuring WebLogic Server Environments*.

The number of `ListenAddress:ListenPort` combinations included in the cluster address is governed by the value of the `NumberOfServersInClusterAddress` attribute on the `ClusterMBean`, which is 3 by default.

You can modify the value of `NumberOfServersInClusterAddress` on the Environments—>Clusters—>*ClusterName*->Configuration->General page of the Administration Console.

- If there are *fewer* Managed Servers available in the cluster than the value of `NumberOfServersInClusterAddress`, the dynamically generated cluster address contains a `ListenAddress:ListenPort` combination for each of the running Managed Servers.
- If there are *more* Managed Servers available in the cluster than the value of `NumberOfServersInClusterAddress`, WebLogic Server randomly selects a subset of the available instances—equal to the value of `NumberOfServersInClusterAddress`—and uses the `ListenAddress:ListenPort` combination for those instances to form the cluster address.

The order in which the `ListenAddress:ListenPort` combinations appear in the cluster address is random—from request to request, the order will vary.

Explicitly Defining Cluster Address for Production Environments

If you explicitly define a cluster address for a cluster in a production environment, specify the cluster address as a DNS name that maps to the IP addresses or DNS names of each WebLogic Server instance in the cluster.

If you define the cluster address as a DNS name, the Listen Ports for the cluster members are not specified in the cluster address—it is assumed that each Managed Server in the cluster has the

BETA

same Listen Port number. Because each server instance in a cluster must have a unique combination of address and Listen Port, if a cluster address is a DNS name, each server instance in the cluster must have:

- a unique address and
- the same Listen Port number

When clients obtain an initial JNDI context by supplying the cluster DNS name, `weblogic.jndi.WLInitialContextFactory` obtains the list of all addresses that are mapped to the DNS name. This list is cached by WebLogic Server instances, and new initial context requests are fulfilled using addresses in the cached list with a round-robin algorithm. If a server instance in the cached list is unavailable, it is removed from the list. The address list is refreshed from the DNS service only if the server instance is unable to reach any address in its cache.

Using a cached list of addresses avoids certain problems with relying on DNS round-robin alone. For example, DNS round-robin continues using all addresses that have been mapped to the domain name, regardless of whether or not the addresses are reachable. By caching the address list, WebLogic Server can remove addresses that are unreachable, so that connection failures aren't repeated with new initial context requests.

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 8-13.

Explicitly Defining Cluster Address for Development and Test Environments

If you explicitly define a cluster address for use in development environments, you can use a cluster DNS name for the cluster address, as described in the previous section.

Alternatively, you can define the cluster address as a list that contains the DNS name (or IP address) and Listen Port of each Managed Server in the cluster, as shown in the examples below:

```
DNSName1:port1,DNSName1:port2,DNSName1:port3
```

```
IPaddress1:port1,IPaddress2:port2;IPaddress3:port3
```

Note that each cluster member has a unique address and port combination.

Explicitly Defining Cluster Address for Single, Multihomed Machine

If your cluster runs on a single, multihomed machine, and each server instance in the cluster uses a different IP address, define the cluster address using a DNS name that maps to the IP addresses of the server instances in the cluster. If you define the cluster address as a DNS name, specify the same Listen Port number for each of the Managed Servers in the cluster.

BETA

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

This section lists typical cluster implementation tasks, and highlights key configuration considerations. The exact process you follow is driven by the unique characteristics of your environment and the nature of your application. These tasks are described:

1. "Install WebLogic Server" on page 7-10
2. "Create a Clustered Domain" on page 7-10
3. "Configure Node Manager" on page 7-12
4. "Configure Load Balancing Method for EJBs and RMIs" on page 7-12
5. "Configure Server Affinity for Distributed JMS Destinations" on page 7-13
6. "Configuring Load Balancers that Support Passive Cookie Persistence" on page 7-13
7. "Configure Proxy Plug-Ins" on page 7-14
8. "Configure Replication Groups" on page 7-21
9. "Configure Migratable Targets for Pinned Services" on page 7-22
10. "Configure Clustered JDBC" on page 7-22
11. "Package Applications for Deployment" on page 7-24
12. "Deploy Applications" on page 7-24
13. "Deploying, Activating, and Migrating Migratable Services" on page 7-26
14. "Configure In-Memory HTTP Replication" on page 7-30
15. "Additional Configuration Topics" on page 7-31

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

BETA

Install WebLogic Server

If you have not already done so, install WebLogic Server. For instructions, see *Installing WebLogic Platform*.

- If the cluster will run on a single machine, do a single installation of WebLogic Server under the `/bea` directory to use for all clustered instances.
- For remote, networked machines, install the same version of WebLogic Server on each machine. Each machine:
  - Must have permanently assigned, static IP addresses. You cannot use dynamically-assigned IP addresses in a clustering environment.
  - Must be accessible to clients. If the server instances are behind a firewall and the clients are in front of the firewall, each server instance must have a public IP address that can be reached by the clients.
  - Must be located on the same local area network (LAN) and must be reachable via IP multicast.

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 server instances must compete to access the filesystem (and possibly to write individual log files). Moreover, should the shared filesystem fail, you might be unable to start clustered server instances.

Create a Clustered Domain

The are multiple methods of creating a clustered domain. For a list, see "Methods of Configuring Clusters" on page 3-8.

For instructions to create a cluster using the:

- Configuration Wizard, first see "Creating a New WebLogic Domain" in *Creating WebLogic Domains Using the Configuration Wizard* for instructions on creating the domain, and then "Customizing your Domain" for instructions on configuring a cluster.

Starting a WebLogic Server Cluster

There are multiple methods of starting a cluster—available options include the command line interface, scripts that contain the necessary commands, and Node Manager.

Note: Node Manager eases the process of starting local and remote Managed Servers, and restarting them after failure. (You cannot use Node Manager to start an Administration

BETA

Server.) To use Node Manager, you must first configure a Node Manager process on each machine that hosts Managed Servers in the cluster. See "Configure Node Manager" on page 7-12.

Regardless of the method you use to start a cluster, start the Administration Server first, then start the Managed Servers in the cluster.

Follow the instructions below to start the cluster from a command shell. Note that each server instance is started in a separate command shell.

1. Open a command shell.
2. Change directory to the domain directory that you created with the Configuration Wizard.
3. Type this command to start the Administration Server:

   ```
   StartWebLogic
   ```

4. Enter the user name for the domain at the "Enter username to boot WebLogic Server" prompt.
5. Enter the password for the domain at the "Enter password to boot WebLogic Server" prompt.

   The command shell displays messages that report the status of the startup process.
6. Open another command shell so that you can start a Managed Server.
7. Change directory to the domain directory that you created with the Configuration Wizard.
8. Type this command

   `StartManagedWebLogic` *`server_name address:port`*

   where:

   server_name is the name of the Managed Server you wish to start

   address is the IP address or DNS name for the Administration Server for the domain

   port is the listen port for the Administration Server for the domain
9. Enter the user name for the domain at the "Enter username to boot WebLogic Server" prompt.
10. Enter the password for the domain at the "Enter password to boot WebLogic Server" prompt.

    The command shell displays messages that report the status of the startup process.

BETA

Note: After you start a Managed Server, it listens for heartbeats from other running server instances in the cluster. The Managed Server builds its local copy of the cluster-wide JNDI tree, as described in "How WebLogic Server Updates the JNDI Tree" on page 2-12, and displays status messages when it has synchronized with each running Managed Server in the cluster. The synchronization process can take a minute or so.

11. To start another server instance in the cluster, return to step 6. Continue through step 10.
12. When you have started all Managed Servers in the cluster, the cluster startup process is complete.

Configure Node Manager

Node Manager is a standalone Java program provided with WebLogic Server that is useful for starting a Managed Server that resides on a different machine than its Administration Server. Node Manager also provides features that help increase the availability of Managed Servers in your cluster. For more information, and for instructions to configure and use Node Manager, see Using Node Manager to Control Servers in *Designing and Configuring WebLogic Server Environments*.

Configure Load Balancing Method for EJBs and RMIs

Follow the instructions in this section to select the load balancing algorithm for EJBs and RMI objects.

Unless you explicitly specify otherwise, WebLogic Server uses the round-robin algorithm as the default load balancing strategy for clustered object stubs. To understand alternative load balancing algorithms, see "Load Balancing for EJBs and RMI Objects" on page 4-4. To change the default load balancing algorithm:

1. Open the WebLogic Server Console.
2. Select the Environments—>Clusters node.
3. Click on the name of your cluster in the table.
4. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
5. Enter the desired load balancing algorithm in the Default Load Algorithm field.
6. Click the Advanced link.

BETA

7. Enter the desired value in the Service Age Threshold field.
8. Click Save to save your changes.
9. Click the Activate Changes button in the top left corner once you are ready to activate your changes.

Configure Server Affinity for Distributed JMS Destinations

To understand the server affinity support provided by WebLogic Server for JMS, see "Load Balancing for JMS" on page 4-14.

Configuring Load Balancers that Support Passive Cookie Persistence

Load balancers that support passive cookie persistence can use information from the WebLogic Server session cookie to associate a client with the WebLogic Server instance that hosts the session. The session cookie contains a string that the load balancer uses to identify the primary server instance for the session.

For a discussion of external load balancers, session cookie persistence, and the WebLogic Server session cookie, see "Load Balancing HTTP Sessions with an External Load Balancer" on page 4-2

To configure the load balancer to work with your cluster, use the facilities of the load balancer to define the offset and length of the string constant.

Assuming that the Session ID portion of the session cookie is the default length of 52 bytes, on the load balancer, set:

- string offset to 53 bytes, the default Random Session ID length plus 1 byte for the delimiter character.
- string length to 10 bytes

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

BETA

Notes: For vendor-specific instructions for configuring Big-IP load balancers, see Appendix B, "Configuring BIG-IP™ Hardware with Clusters."

Configure Proxy Plug-Ins

Refer to the instructions in this section if you wish to load balance servlets and JSPs using a proxy plug-in. A proxy plug-in proxies requests from a web server to WebLogic Server instances in a cluster, and provides load balancing and failover for the proxied HTTP requests.

For information about load balancing using proxy plug-ins, see "Load Balancing with a Proxy Plug-in" on page 4-2. For information about connection and failover using proxy plug-ins, see "Replication and Failover for Servlets and JSPs" on page 5-2, and "Accessing Clustered Servlets and JSPs Using a Proxy" on page 5-8.

- If you use WebLogic Server as a web server, set up `HttpClusterServlet` using the instructions in "Set Up the HttpClusterServlet" on page 7-14.
- If you use a supported third-party web server, set up a product-specific plug-in (for a list of supported web servers, see "Load Balancing with a Proxy Plug-in" on page 4-2.), follow the instructions in *Using WebLogic Server with Plug-ins*.

Note: Each web server that proxies requests to a cluster must have an identically configured plug-in.

Set Up the HttpClusterServlet

To use the HTTP cluster servlet, configure it as the default web application on your proxy server machine, as described in the steps below. For an introduction to web applications, see "Overview of Web Applications" in *Developing Web Applications for WebLogic Server*.

1. If you have not already done so, configure a separate, non-clustered Managed Server to host the HTTP Cluster Servlet.
2. Create the `web.xml` deployment descriptor file for the servlet. This file must reside in the `\WEB-INF` subdirectory of the web application directory. A sample deployment descriptor for the proxy servlet is provided in "Sample web.xml" on page 7-15. For more information on `web.xml`, see "Understanding Web Applications, Servlets, and JSPs" in *Developing Web Applications, Servlets, and JSPs for WebLogic Server.*
   a. Define the name and class for the servlet in the `<servlet>` element in `web.xml`. The servlet name is `HttpClusterServlet`. The servlet class is `weblogic.servlet.proxy.HttpClusterServlet`.

BETA

b. Identify the clustered server instances to which the proxy servlet will direct requests in the `<servlet>` element in `web.xml`, by defining the `WebLogicCluster` parameter.

c. Create `<servlet-mapping>` stanzas to specify the requests that the servlet will proxy to the cluster, using the `<url-pattern>` element to identify specific file extensions, for example `*.jsp`, or `*.html`. Define each pattern in a separate `<servlet-mapping>` stanza.

   You can set the `<url-pattern>` to "`/`" to proxy any request that cannot be resolved by WebLogic Server to the remote server instance. If you do so, you must also specifically map the following extensions: `*.jsp`, `*.html`, and `*.html`, to proxy files ending with those extensions. For an example, see "Sample web.xml" on page 7-15.

d. Define, as appropriate, any additional parameters. See Table 7-1 for a list of key parameters. See "Parameters for Web Server Plug-ins" in *Using WebLogic Server with Plug-ins* for a complete list. Follow the syntax instructions in "Proxy Servlet Deployment Parameters" on page 7-17.

3. Create the `weblogic.xml` deployment descriptor file for the servlet. This file must reside in the `\WEB-INF` subdirectory of the web application directory.

   Assign the proxy servlet as the default web application for the Managed Server on the proxy machine by setting the `<context-root>` element to a forward slash character (/) in the `<weblogic-web-app>` stanza. For an example, see "Sample weblogic.xml" on page 7-17.

4. In the Administration Console, deploy the servlet to the Managed Server on your proxy server machine. For instructions, see "Deploying a New Web Application" in *Administration Console Online Help.*

Sample web.xml

This section contains a sample deployment descriptor file (`web.xml`) for `HttpClusterServlet`.

`web.xml` defines parameters that specify the location and behavior of the proxy servlet: both versions of the servlet:

- The `DOCTYPE` stanza specifies the DTD used by WebLogic Server to validate `web.xml`.
- The `servlet` stanza:
  - Specifies the location of the proxy plug-in servlet class. The file is located in the `weblogic.jar` in your `WL_HOME/server/lib` directory. You do not have to specify the servlet's full directory path in `web.xml` because `weblogic.jar` is put in your CLASSPATH when you start WebLogic Server.

BETA

- Identifies the host name and listen port of each Managed Servers in the cluster, using the `WebLogicCluster` parameter.

- The three `servlet-mapping` stanzas specify that the servlet will proxy URLs that end in '/', 'htm', 'html', or 'jsp' to the cluster.

For parameter definitions see "Proxy Servlet Deployment Parameters" on page 7-17.

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd";>

<web-app>

<servlet>
  <servlet-name>HttpClusterServlet</servlet-name>
    <servlet-class>
      weblogic.servlet.proxy.HttpClusterServlet
    </servlet-class>

  <init-param>
    <param-name>WebLogicCluster</param-name>
    <param-value>
       myserver1:7736|myserver2:7736|myserver:7736
    </param-value>
  </init-param>

</servlet>

<servlet-mapping>
  <servlet-name>HttpClusterServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>HttpClusterServlet</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>HttpClusterServlet</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>HttpClusterServlet</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>

</web-app>
```

BETA

Sample weblogic.xml

This section contains a sample `weblogic.xml` file. The `<context-root>` deployment parameter is set to "/". This makes the proxy servlet the default web application for the proxy server.

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 8.1//EN" "http://www.bea.com/servers/wls810/dtd/weblogic
810-web-jar.dtd">
  <weblogic-web-app>
    <context-root>/</context-root>
  </weblogic-web-app>
```

Proxy Servlet Deployment Parameters

Key parameters for configuring the behavior of the proxy servlet in `web.xml` are listed in Table 7-1.

The parameters for the proxy servlet are the same as those used to configure WebLogic Server plug-ins for Apache, Microsoft, and Netscape web servers. For a complete list of parameters for configuring the proxy servlet and the plug-ins for third-part web servers see "Parameters for Web Server Plug-ins" in *Using WebLogic Server with Plug-ins*.

The syntax for specifying the parameters, and the file where they are specified, is different for the proxy servlet and for each of the plug-ins.

For the proxy servlet, specify the parameters in `web.xml`, each in its own `<init-param>` stanza within the `<servlet>` stanza of `web.xml`. For example:

```
<init-param>
    <param-name>ParameterName</param-name>
    <param-value>ParameterValue</param-value>
</init-param>
```

BETA

Table 7-1 Proxy Servlet Deployment Parameters

Parameter	Usage

`WebLogicCluster`	`<init-param>`
	`<param-name>WebLogicCluster</param-name>`
	`<param-value>WLS1.com:`*`port`*`|WLS2.com:`*`port`*
	`</param-value>`
	Where `WLS1.com` and `WLS2.com` are the host names of servers in the cluster, and *`port`* is a port where the host is listening for HTTP requests.
	If you are using SSL between the plug-in and WebLogic Server, set the port number to the SSL listen port (see Configuring the Listen Port) and set the `SecureProxy` parameter to ON.

`SecureProxy`	`<init-param>`
	`<param-name>SecureProxy</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`
	Valid values are ON and OFF.
	If you are using SSL between the plug-in and WebLogic Server, set the port number to the SSL listen port (see Configuring the Listen Port) and set the `SecureProxy` parameter to ON.

`DebugConfigInfo`	`<init-param>`
	`<param-name>DebugConfigInfo</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`
	Valid values are ON and OFF.
	If set to ON, you can query the `HttpClusterServlet` for debugging information by adding a request parameter of `?__WebLogicBridgeConfig` to any request. (Note: There are two underscore (_) characters after the ?.) For security reasons, it is recommended that you set the `DebugConfigInfo` parameter to OFF in a production environment.

BETA

Parameter	Usage

`ConnectRetrySecs`	Interval in seconds that the servlet will sleep between attempts to connect to a server instance. Assign a value less than `ConnectTimeoutSecs`.
	The number of connection attempts the servlet makes before returning an `HTTP 503/Service Unavailable` response to the client is `ConnectTimeoutSecs` divided by `ConnectRetrySecs`.
	Syntax:
	`<init-param>`
	`<param-name>ConnectRetrySecs</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`

`ConnectTimeoutSecs`	Maximum time in seconds that the servlet will attempt to connect to a server instance. Assign a value greater than `ConnectRetrySecs`.
	If `ConnectTimeoutSecs` expires before a successful connection, an `HTTP 503/Service Unavailable` response is sent to the client.
	Syntax:
	`<init-param>`
	`<param-name>ConnectTimeoutSecs</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`

`PathTrim`	String trimmed by the plug-in from the beginning of the original URL, before the request is forwarded to the cluster.
	Syntax:
	`<init-param>`
	`<param-name>PathTrim</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`
	Example:
	If the URL
	`http://myWeb.server.com/weblogic/foo`
	is passed to the plug-in for parsing and if `PathTrim` has been set to
	`/weblogic`
	the URL forwarded to WebLogic Server is:
	`http://myWeb.server.com:7001/foo`

BETA

Parameter	Usage

`TrimExt`	The file extension to be trimmed from the end of the URL.
	Syntax:
	`<init-param>`
	`<param-name>TrimExt</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`

`clientCertProxy`	Specifies to trust client certificates in the `WL-Proxy-Client-Cert` header. Valid values are true and false. The default value is false. This setting is useful if user authentication is performed on the proxy server—setting `clientCertProxy` to true causes the proxy server to pass on the certs to the cluster in a special header, `WL-Proxy-Client-Cert`. The `WL-Proxy-Client-Cert` header can be used by any client with direct access to WebLogic Server. WebLogic Server takes the certificate information from that header, trusting that is came from a secure source (the plug-in) and uses that information to authenticate the user. For this reason, if you set `clientCertProxy` to true, use a connection filter to ensure that WebLogic Server accepts connections only from the machine on which the plug-in is running. See "Using Network Connection Filters" in *Programming WebLogic Security*.

`PathPrepend`	String that the servlet prepends to the original URL, after `PathTrim` is trimmed, before forwarding the URL to the cluster.
	`<init-param>`
	`<param-name>PathPrepend</param-name>`
	`<param-value>`*`ParameterValue`*`</param-value>`
	`</init-param>`

BETA

Accessing Applications Via the Proxy Server

Ensure that applications clients will access via the proxy server are deployed to your cluster. Address client requests to the listen address and listen port of the proxy server.

If you have problems:

- Make sure all servers instances have unique address/port combinations

  Each of the server instances in the configuration must have a unique combination of Listen Address and Listen Port.

- Make sure that the proxy servlet is the default application for the proxy server

If you get a page not found error when you try to your application, make sure that `weblogic.xml` is in `\WEB-INF` for the application and that it sets the `context-root` deployment parameter to "/".

- When all else fails, restart

  If you are having problems try rebooting all your servers, some of the changes you made while configuring your setup may not have been persisted to the configuration file.

- Verify Your Configuration

  To verify the configuration of the `HttpClusterServlet`:

  a. Set the `DebugConfigInfo` parameter in `web.xml` to ON.

  b. Use a Web browser to access the following URL:

     `http://myServer:port/placeholder.jsp?__WebLogicBridgeConfig`

     Where:

     `myServer` is the Managed Server on the proxy machine where `HttpClusterServlet` runs,

     port is the port number on that server that is listening for HTTP requests, and

     `placeholder.jsp` is a file that does not exist on the server.

     The plug-in gathers configuration information and run-time statistics and returns the information to the browser. For more information, see "Parameters for Web Server Plug-ins" in *Using WebLogic Server with Plug-ins*.

Configure Replication Groups

To support automatic failover for servlets and JSPs, WebLogic Server replicates HTTP session states in memory. You can further control where secondary states are placed using *replication groups*. A replication group is a preferred list of clustered instances to be used for storing session state replicas.

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.

For instructions on how to determine which server instances should participate in each replication group, and to determine each server instance's preferred replication group, follow the instructions in "Using Replication Groups" on page 5-5.

Then follow these steps to configure replication groups for each WebLogic Server instance:

BETA

To configure replication groups for a WebLogic Server instance:

1. Open the WebLogic Server Console.
2. Select the Environments->Servers node.
3. In the table, click on the name of the server you want to configure.
4. Select the Cluster tab.
5. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
6. Enter values for the following attribute fields:
   - Replication Group: Enter the name of the replication group 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.
7. Click Save to save your changes.
8. Click the Activate Changes button in the top left corner to activate your changes.

Configure Migratable Targets for Pinned Services

WebLogic Server enables you to configure an optional migratable target, which defines a list of server instances in the cluster that can potentially host a migratable service, such as a JMS server or the Java Transaction API (JTA) transaction recovery service. If you want to use a migratable target, configure the target server list before deploying or activating the service in the cluster.

If you do not configure a migratable target in the cluster, migratable services can be migrated to any WebLogic Server instance in the cluster. See "Migration for Singleton Services" on page 5-31 for more information.

For instructions on configuring migratable JMS targets, see "Configuring Clustered WebLogic JMS Resources" in *Configuring and Managing WebLogic JMS.*

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.

BETA

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 1-6.
- For a description of how clustered JDBC can increase application availability, see "Failover and JDBC Connections" on page 5-34.
- For a description of how clustered JDBC supports load balancing, see "Load Balancing for JDBC Connections" on page 4-16.

Clustering Connection Pools

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

1. Create a connection pool.
2. Assign the connection pool to the cluster.
3. Create the data source. Specify the connection pool created in the previous step in the `Pool Name` attribute.
4. Assign the data source to the cluster.

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 1-6.

1. Create two or more connection pools.
2. Assign each connection pool to the cluster.
3. Create a multipool. Assign the connection pools created in the previous step to the multipool.
4. Assign the Multipool to the cluster.

BETA

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

Package Applications for Deployment

You must package applications before you deploy them to WebLogic Server. For more information, see the packaging topic in "Deploying the Application" in *Developing Applications for WebLogic Server*.

Deploy Applications

Clustered objects in WebLogic Server should be deployed homogeneously. To ensure homogeneous deployment, when you select a target use the cluster name, rather than individual WebLogic Server instances in the cluster.

The console automates deploying replica-aware objects to clusters. When you deploy an application or object to a cluster, the 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.) For a discussion of application deployment in clustered environments see "Application Deployment Topics" on page 3-4. For a broad discussion of deployment topics, see *Deploying WebLogic Server Applications*.

Note: All server instances in your cluster should be running when you deploy applications to the cluster using the Administration Console

Deploying to a Single Server Instance (Pinned Deployment)

Deploying a application to a server instance, rather than the all cluster members is called a pinned deployment. Although a pinned deployment targets a specific server instance, all server instances in the cluster must be running during the deployment process.

You can perform a pinned deployment using the Administration Console or from the command line, using `weblogic.Deployer`.

Pinned Deployment from the Command Line

From a command shell, use the following syntax to target a server instance:

```
java weblogic.Deployer -activate -name ArchivedEarJar -source
C:/MyApps/JarEar.ear -target server1
```

BETA

Cancelling Cluster Deployments

You can cancel a deployment using the Administration Console or from the command line, using `weblogic.Deployer`.

Cancel Deployment from the Command Line

From a command shell, use the following syntax to cancel the deployment task ID:

```
java weblogic.Deployer -adminurl http://admin:7001 -cancel -id tag
```

Cancel Deployment Using the Administration Console

In the Administration Console, open the Tasks node to view and to cancel any current deployment tasks.

Viewing Deployed Applications

To view a deployed application in the Administration Console:

1. In the Console, click the Deployments node.
2. View a list of deployed applications in the table displayed in the Console.

Undeploying Deployed Applications

To undeploy a deployed application from the WebLogic Server Administration Console:

1. In the Console, click Deployments.
2. In the displayed table, check the checkbox to the left of the application you want to undeploy.
3. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
4. Click Stop.
5. Select when you want the application to stop (undeploy).
6. Click Yes.
7. Click the Activate Changes button in the top left corner of the console to activate your changes.

BETA

Deploying, Activating, and Migrating Migratable Services

The sections that follow provide guidelines and instructions for deploying, activating, and migrating migratable services. For a discussion of migratable services, see "Migration for Singleton Services" on page 5-31

Deploying JMS to a Migratable Target Server Instance

The migratable target that you create defines the scope of server instances in the cluster that can potentially host a migratable service. You must deploy or activate a pinned service on one of the server instances listed in the migratable target in order to migrate the service within the target server list at a later time. Use the instructions that follow to deploy a JMS server on a migratable target, or activate the JTA transaction recovery system so that you can migrate it later.

Note: If you did not configure a migratable target, simply deploy the JMS server to any WebLogic Server instance in the cluster; you can then migrate the JMS server to any other server instance in the cluster (no migratable target is used).

To deploy a JMS server to a migratable target using the Administration Console:

1. Use the instructions in "Configure Migratable Targets for Pinned Services" on page 7-22 to create a migratable target for the cluster, if you have not already done so.
2. Start the Administration Server for the cluster and log in to the Administration Console.
3. If you have not already done so, click the Lock and & Edit button in the top left corner of the console.
4. Select the Services->JMS->Servers node in the left pane.
5. In the displayed table, click the name of the configured JMS server that you want to deploy to the cluster. This displays the JMS server configuration in the right pane.
6. Select the Targets and Deploy tab in the right pane.
7. Select the name of a server instance from the drop-down list. The drop-down list specifies server names that are defined as part of a migratable target.
8. Click Save to save your changes.
9. Click the Activate Changes button in the top left corner of the console to activate your changes.

BETA

Activating JTA as a Migratable Service

The JTA recovery service is automatically started on one of the server instances listed in the migratable target for the cluster; you do not have to deploy the service to a selected server instance.

If you did not configure a JTA migratable target, WebLogic Server activates the service on any available WebLogic Server instance in the cluster. To change the current server instance that hosts the JTA service, use the instructions in "Migrating a Pinned Service to a Target Server Instance" on page 7-27.

Migrating a Pinned Service to a Target Server Instance

After you have deployed a migratable service, you can use the Administration Console to migrate the service to another server instance in the cluster. If you configured a migratable target for the service, you can migrate to any other server instance listed in the migratable target, even if that server instance is not currently running. If you did not configure a migratable target, you can migrate the service to any other server instance in the cluster.

If you migrate a service to a stopped server instance, the server instance will activate the service upon the next startup. If you migrate a service to a running WebLogic Server instance, the migration takes place immediately.

To migrate a pinned service using the Administration Console:

1. Use the instructions in "Deploying JMS to a Migratable Target Server Instance" on page 7-26 to deploy a pinned service to the cluster, if you have not already done so.
2. Start the Administration Server for the cluster and log in to the Administration Console.
3. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
4. Select the Environments->Servers node in the left pane.
5. Click on the name of the server instance that is a member of the cluster you want to configure.
6. Select Control->Migrate if you want to migrate the JMS service, or Control->JTA Migrate to migrate the JTA transaction recovery service.

   The Current Server field shows the WebLogic Server instance that currently hosts the pinned service. The Destination Server drop-down list displays server instances to which you can migrate the service.

BETA

7. Use the Destination Server drop-down list to select the new server instance that will host the pinned service.
8. Click Migrate to migrate the pinned service from the Current Server to the Destination Server.
9. If the Current Server is not reachable by the Administration Server, the Administration Console displays this message:

   ```
   Unable to contact server MyServer-1, the source server from which
   services are being migrated.

   Please ensure that server MyServer-1 is NOT running! If the
   administration server cannot reach server MyServer-1 due to a network
   partition, inspect the server directly to verify that it is not
   running. Continue the migration only if MyServer-1 is not running.
   Cancel the migration if MyServer-1 is running, or if you do not know
   whether it is running.

   Before bringing up MyServer-1 again after this migration, use the java
   weblogic.PurgeConfigCache utility to prevent redundant activation of
   the migrated service. See Help for more information
   ```

   If this message is displayed, perform the procedure described in "Migrating When the Currently Active Host is Unavailable" on page 7-28.
10. If the Destination Server is stopped, the Administration Console notifies you of the stopped server instance and asks if you would like to continue the migration. Click the Continue button to migrate to the stopped server instance, or click Cancel to stop the migration and select a different server instance.
11. The migration process may take several minutes to complete, depending on the server instance configuration. However, you can continue using other Administration Console features while the migration takes place. To view the migration status at a later time, click the Tasks node in the left pane to display the currently-running tasks for the domain; then select the task description for the migration task to view the current status.

Migrating When the Currently Active Host is Unavailable

Use this migration procedure if a clustered Managed Server that was the active server for the migratable service crashes or becomes unreachable.

This procedure purges the failed Managed Server's configuration cache. The purpose of purging the cache is to ensure that, when the failed server instance is once again available, it does not re-deploy a service that you have since migrated to another Managed Server. Purging the cache

BETA

eliminates the risk that Managed Server which was previously the active host for the service uses local, out-of-date configuration data when it starts up again.

1. Disconnect the machine from the network entirely. It should not be accessible to the Administration Server or client traffic. If the machine has a dual ported disk, disconnect it.
2. Migrate the migratable service(s) to a Managed Server instance on a different machine. The Administration Server must be running, so that it can coordinate the migration and update the activation table.
   - If you use the command line for migration, use the `-sourcedown` flag.
   - If you use the console, it will ask you to make sure the source server is not going to restart.

   The migratable service is now available on a different Managed Server on a different machine. The following steps can be performed at leisure.
3. Perform the necessary repair or maintenance on the failed machine.
4. Reboot the machine, but do not connect it to the network.

   Node Manager will start as a service or daemon, and will attempt to start the Managed Servers on the machine.
   - If Managed Server Independence is enabled, the Managed Server will start, even though it cannot connect to the Administration Server.
   - If Managed Server Independence is disabled, the Managed Server will not start, because it cannot connect to the Administration Server.
5. Use the `java weblogic.PurgeConfigCache` utility to disable all Managed Servers that host migratable services on the machine. Run the utility from the WebLogic Server home directory on the machine. If there are multiple installations on the machine, each with different root directories, participating in the cluster, run the utility from each WebLogic Server home directory.

   The utility will:
   - Remove the Managed Servers' PIDs from the monitor process list—the list of server instances to be restarted after failure.
   - Shut down any Managed Servers that started when the machine was rebooted, and prevent them from restarting again.
   - Delete the replicated `config.xml` from the Managed Server's root directory.

BETA

6. Reconnect the machine to the network and shared storage, including dual ported disk, if applicable.
7. Restart the Node Manager daemon/service or reboot the machine, to start all remaining Managed Servers.
8. Start the Managed Serves that was disabled in step 5. This is a normal start up, rather than a restart performed by Node Manager. The Administration Server must be reachable and running, so that the Managed Servers can synchronize with the migratable service activation table on the Administration Server—and hence know that it is no longer the active host of the migratable service.

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 "Using Sessions and Session Persistence" in *Developing Web Applications, Servets, and JSPs for WebLogic Server*.

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>
```

BETA

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 2-4, and "Client Communication via Sockets" on page 2-8.
- For instructions on how to determine how many socket reader threads are necessary in your cluster, see "Determining Potential Socket Usage" on page 2-6. 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 Considerations for Multi-Tier Architecture" on page 6-10.

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 Administration Console.
2. Select the Environments->Servers node.
3. Click the name of the server instance you want to configure.
4. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
5. Select the Configuration->Tuning tab.
6. Check the Enable Native IO box.
7. Click Save.

BETA

8. Click the Activate Changes button in the top left corner of the console to activate your 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 Administration Console.
2. Select the Environments->Servers node.
3. Click the name of the server instance you want to configure.
4. If you have not already done so, click the Lock & Edit button in the top left corner of the console.
5. Select the Configuration->Tuning tab.
6. Edit the percentage of Java reader threads in the Socket Readers 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 field).
7. Click Save to save your changes.
8. Click the Activate Changes button in the top left corner of the console to activate your changes.

Set the Number of Reader Threads on Client Machines

On client machines, you can configure the number 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

BETA

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 "If Your Cluster Spans Multiple Subnets in a WAN" on page 2-2.

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&.

Configure Machine Names

Configure a Machine Name if:

BETA

- Your cluster will span multiple machines, and multiple server instances will run on individual machines in the cluster, or
- You plan to run Node Manager on a machine that does not host a Administration Server

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 machines that host multiple 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 server instances to host secondary HTTP session state replicas, as described in "Using Replication Groups" on page 5-5.

Configuration Notes for Multi-Tier Architecture

If your cluster has a multi-tier architecture, see the configuration guidelines in "Configuration Considerations for Multi-Tier Architecture" on page 6-10.

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 *Developing Web Applications for WebLogic Server*.

BETA

CHAPTER 8

Clustering Best Practices

BETA

These topics recommend design and deployment practices that maximize the scalability, reliability, and performance of applications hosted by a WebLogic Server Cluster.

General Design Considerations

The following sections describe general design guidelines for clustered applications.

Strive for Simplicity

Distributed systems are complicated by nature. For a variety of reasons, make simplicity a primary design goal. Minimize "moving parts" and do not distribute algorithms across multiple objects.

Minimize Remote Calls

You improve performance and reduce the effects of failures by minimizing remote calls.

Session Facades Reduce Remote Calls

Avoid accessing EJB entity beans from client or servlet code. Instead, use a session bean, referred to as a *facade*, to contain complex interactions and reduce calls from web applications to RMI objects. When a client application accesses an entity bean directly, each getter method is a remote call. A session facade bean can access the entity bean locally, collect the data in a structure, and return it by value.

Transfer Objects Reduce Remote Calls

EJBs consume significant system resources and network bandwidth to execute—they are unlikely to be the appropriate implementation for every object in an application.

Use EJBs to model logical groupings of an information and associated business logic. For example, use an EJB to model a logical subset of the line items on an invoice—for instance, items to which discounts, rebates, taxes, or other adjustments apply.

In contrast, an individual line item in an invoice is fine-grained—implementing it as an EJB wastes network resources. Implement objects that simply represents a set of data fields, which require only `get` and `set` functionality, as *transfer objects*.

Transfer objects (sometimes referred to as *value objects* or *helper classes*) are good for modeling entities that contain a group of attributes that are always accessed together. A transfer object is a serializable class within an EJB that groups related attributes, forming a composite value. This class is used as the return type of a remote business method.

Clients receive instances of this class by calling coarse-grained business methods, and then locally access the fine-grained values within the transfer object. Fetching multiple values in one server round-trip decreases network traffic and minimizes latency and server resource usage.

Distributed Transactions Increase Remote Calls

Avoid transactions that span multiple server instances. Distributed transactions issue remote calls and consume network bandwidth and overhead for resource coordination.

Web Application Design Considerations

The following sections describe design considerations for clustered servlets and JSPs.

BETA

Configure In-Memory Replication

To enable automatic failover of servlets and JSPs, session state must persist in memory. For instructions to configure in-memory replication for HTTP session states, see "Requirements for HTTP Session State Replication" on page 5-3 and "Configure In-Memory HTTP Replication" on page 7-30.

Design for Idempotence

Failures or impatient users can result in duplicate servlet requests. Design servlets to tolerate duplicate requests.

Programming Considerations

See "Programming Considerations for Clustered Servlets and JSPs" on page 5-4.

EJB Design Considerations

The following sections describe design considerations for clustered RMI objects.

Design Idempotent Methods

It is not always possible to determine when a server instance failed with respect to the work it was doing at the time of failure. For instance, if a server instance fails after handling a client request but before returning the response, there is no way to tell that the request was handled. A user that does not get a response retries, resulting in an additional request.

Failover for RMI objects requires that methods be *idempotent*. An idempotent method is one that can be repeated with no negative side-effects.

Follow Usage and Configuration Guidelines

The following table summarizes usage and configuration guidelines for EJBs. For a list of configurable cluster behaviors, see Table 8-2 on page 6.

BETA

Table 8-1 EJB Types and GuidelinesS

Object Type	Usage	Configuration
EJBs of all types	Use EJBs to model logical groupings of an information and associated business logic. See "Transfer Objects Reduce Remote Calls" on page 8-2.	Configure clusterable homes
Stateful session beans	Recommended for high volume, heavy-write transactions. Remove stateful session beans when finished to minimize EJB container overhead. A stateful session bean instance is associated with a particular client, and remains in the container until explicitly removed by the client, or removed by the container when it times out. Meanwhile, the container may passivate inactive instances to disk. This consumes overhead and can affect performance. **Note:** Although unlikely, the current state of a stateful session bean can be lost. For example, if a client commits a transaction involving the bean 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 bean. If it is critical to preserve bean state in all possible failover scenarios, use an entity EJB rather than a stateful session EJB.	Configure clusterable homes Configure in-memory replication for EJBs
Stateless Session Beans	Scale better than stateful session beans which are instantiated on a per client basis, and can multiply and consume resources rapidly. When a home creates a stateless bean, it returns a replica-aware stub that can route to any server where 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.	Configure clusterable homes. Configure Cluster Address. Configure methods to be idempotence to support failover during method calls. (Failover is default behavior if failure occurs between method calls.or if the method fails to connect to a server). The methods on stateless session bean homes are automatically set to be idempotent. It is not necessary to explicitly specify them as idempotent.

BETA

Object Type	Usage	Configuration
Read-only Entity Beans	Recommended whenever stale data is tolerable—suitable for product catalogs and the majority of content within many applications. Reads are performed against a local cache that is invalided on a timer basis. Read-only entities perform three to four times faster than transactional entities. **Note:** A client can successfully call setter methods on a read-only entity bean, however the data will never be moved into the persistent store.	Configure clusterable homes. Configure Cluster Address. Methods are configured to be idempotent by default.
Read-Write Entity Beans	Best suited for shared persistent data that is not subject to heavy request and update.If the access/update load is high, consider session beans and JDBC. Recommended for applications that require high data consistency, for instance, customer account maintenance. All reads and writes are performed against the database. Use the `isModified` method to reduce writes. For read-mostly applications, characterized by frequent reads, and occasional updates (for instance, a catalog)—a combination of read-only and read-write beans that extend the read-only beans is suitable. The read-only bean provides fast, weakly consistent reads, while the read-write bean provides strongly consistent writes.	Configure clusterable homes Configure methods to be idempotence to support failover during method calls. (Failover is default behavior if failure occurs between method calls.or if the method fails to connect to a server). The methods on read-only entity beans are automatically set to be idempotent.

Cluster-Related Configuration Options

The following table lists key behaviors that you can configure for a cluster, and the associated method of configuration.

BETA

Table 8-2 Cluster-Related Configuration Options

Configurable Behavior or Resource	How to Configure
clusterable homes	Set `home-is-clusterable` in `weblogic-ejb-jar.xml` to "true.
idempotence	At bean level, set `stateless-bean-methods-are-idempotent` in `weblogic-ejb-jar.xml` to "true". At method level, set `idempotent-methods` in `weblogic-ejb-jar.xml`
in-memory replication for EJBs	Set `replication-type` in `weblogic-ejb-jar.xml` to "InMemory".
Cluster Address	The cluster address 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. The cluster address can be assigned explicitly, or generated automatically by WebLogic Server for each request. For more information, see "Cluster Address" on page 7-6.
`clients-on-same-server`	Set `clients-on-same-server` in `weblogic-ejb-jar.xml` to "True" if all clients that will access the EJB will do so from the same server on which the bean is deployed. If `clients-on-same-server` is "True" the server instance will not multicast JNDI announcements for the EJB when it is deployed, hence reducing the startup time for a large clusters.
Load balancing algorithm for entity bean and entity EJBs homes	`home-load-algorithm` in `weblogic-ejb-jar.xml` specifies the algorithm to use for load balancing between replicas of the EJB home. If this element is not defined, WebLogic Server uses the algorithm specified by the `weblogic.cluster.defaultLoadAlgorithm` attribute in `config.xml`.
Custom load balancing for entity EJBs, stateful session EJBs, and stateless session	Use `home-call-router-class-name` in `weblogic-ejb-jar.xml` to specify the name of a custom class to use for routing bean method calls for these types of beans. This class must implement `weblogic.rmi.cluster.CallRouter()`. For more information, see "The WebLogic Cluster API" on page A-1.

BETA

Configurable Behavior or Resource	How to Configure
Custom load balancing for stateless session bean	Use `stateless-bean-call-router-class-name` in `weblogic-ejb-jar.xml` to specify the name of a custom class to use for routing stateless session bean method calls. This class must implement `weblogic.rmi.cluster.CallRouter()`. For more information, see "The WebLogic Cluster API" on page A-1.
Configure stateless session bean as clusterable	Set `stateless-bean-is-clusterable` in `weblogic-ejb-jar.xml` to "true" to allow the EJB to be deployed to a cluster.
Load balancing algorithm for stateless session beans.	Use `stateless-bean-load-algorithm` in `weblogic-ejb-jar.xml` to specify the algorithm to use for load balancing between replicas of the EJB home. If this property is not defined, WebLogic Server uses the algorithm specified by the `weblogic.cluster.defaultLoadAlgorithm` attribute in `config.xml`.
Machine	The WebLogic Server Machine resource associates server instances with the computer on which it runs. For more information, see "Configure Machine Names" on page 7-33.
Replication groups	Replication groups allow you to control where HTTP session states are replicated. For more information, see "Configure Replication Groups" on page 7-21

BETA

State Management in a Cluster

Different services in a WebLogic Server cluster provide varying types and degrees of state management. This list defines four categories of service that are distinguished by how they maintain state in memory or persistent storage:

- Stateless services—A stateless service does not maintain state in memory between invocations.
- Conversational services—A conversational service is dedicated to a particular client for the duration of a session. During the session, it serves all requests from the client, and only requests from that client. Throughout a session there is generally state information that the the application server must maintain between requests. Conversational services typically maintain transient state in memory, which can be lost in the event of failure. If session state

is written to a shared persistent store between invocations, the service is stateless. If persistent storage of state is not required, alternatives for improving performance and scalability include:

  - Session state can be sent back and forth between the client and server under the covers, again resulting in a stateless service. This approach is not always feasible or desirable, particularly with large amounts of data.
  - More commonly, session state may be retained in memory on the application server between requests. Session state can be paged out from memory as necessary to free up memory. Performance and scalability are still improved in this case because updates are not individually written to disk and the data is not expected to survive server failures.

- Cached services—A cached service maintains state in memory and uses it to process requests from multiple clients. Implementations of cached services vary in the extent to which they keep the copies of cached data consistent with each other and with associated data in the backing store.
- Singleton services—A singleton service is active on exactly one server in the cluster at a time and processes requests from multiple clients. A singleton service is generally backed by private, persistent data, which it caches in memory. It may also maintain transient state in memory, which is either regenerated or lost in the event of failure. Upon failure, a singleton service must be restarted on the same server or migrated to a new server.

Table 8-3, "J2EE and WebLogic Support for Service Types," on page 8-9 summarizes how J2EE and WebLogic support different each of these categories of service.

Note: In Table 8-3, support for stateless and conversational services is described for two types of clients:

- *Loosely-coupled* clients include browsers or Web Service clients that communicate with the application server using standard protocols.
- *Tightly-coupled* clients are objects that run in the application tier or in the client-side environment, and communicate with the application server using proprietary protocols.

BETA

Table 8-3 J2EE and WebLogic Support for Service Types

Service	J2EE Support	WebLogic Server Scalability and Reliability Features for...
Stateless Service with loosely-coupled clients	All J2EE APIs are either stateless or may be implemented in a stateless manner by writing state information to a shared persistent store between invocations. J2EE does not specify a standard for load balancing and failover. For loosely coupled clients, load balancing must be must be performed by external IP-based mechanisms	WebLogic Server increases the availability of stateless services by deploying multiple instances of the service to a cluster. For loosely-coupled clients of a stateless service, WebLogic Server supports external load balancing solutions, and provides proxy plug-ins for session failover and load balancing. For more information, see: • "Stateless Session Beans" on page 5-15 • "Load Balancing HTTP Sessions with an External Load Balancer" on page 4-2 • "Load Balancing with a Proxy Plug-in" on page 4-2
Stateless Service with tightly-coupled clients	These J2EE APIs support tightly coupled access to stateless services: • JNDI (after initial access) • Factories, such as EJB homes, JDBC connection pools, and JMS connection factories • Stateless session beans • Entity beans, if written to a shared persistent store between invocations	WebLogic Server increases the availability of stateless services by deploying multiple instances of the service to a cluster. For tightly-coupled clients of a stateless service, WebLogic Server supports load balancing and failover in its RMI implementation. The WebLogic Server replica-aware stub for a clustered RMI object lists the server instances in the cluster that currently offer the service, and the configured load balancing algorithm for the object. WebLogic Server uses the stub to make load balancing and failover decisions. For more information, see: • "Stateless Session Beans" on page 5-15 • "Load Balancing for EJBs and RMI Objects" on page 4-4

BETA

Service	J2EE Support	WebLogic Server Scalability and Reliability Features for...
Conversational services with loosely-coupled clients	These J2EE APIs support loosely-coupled access to conversational services: • Servlets • Web Services J2EE does not specify a standard for load balancing and failover. Load balancing can be accomplished with external IP-based mechanisms or application server code in the presentation tier. Because protocols for conversations services are stateless, load balancing should occur only when the session is created. Subsequent requests should stick to the selected server.	WebLogic Server increases the reliability of sessions with: • Failover, based on in-memory replication of session state, and distribution of primaries and secondaries across the cluster. • Configurable replication groups, and the ability to specify preferred replication groups for hosting secondaries. • Load balancing using external load balancers or proxy-plug-ins. For more information, see • "HTTP Session State Replication" on page 5-3 • "Load Balancing for Servlets and JSPs" on page 4-1.
Conversational services with tightly-coupled clients	The J2EE standard provides EJB stateful session beans to support conversational services with tightly-coupled clients.	WebLogic Server increases the availability and reliability of stateful session beans with these features: • Caching • Persistent storage of passivated bean state. • Initial load balancing occurs when an EJB home is chosen to create the bean. The replica-aware stub is hard-wired to the chosen server, providing session affinity. • When primary/secondary replication is enabled, the stub keeps track of the secondary and performs failover. • Updates are sent from the primary to the secondary only on transaction boundaries. For more information, see "Stateful Session Beans" on page 5-15.

BETA

Service	J2EE Support	WebLogic Server Scalability and Reliability Features for...
Cached Services	J2EE does not specify a standard for cached services. Entity beans with Bean-Managed-Persistence can implement custom caches.	Weblogic Server supports caching of: **Stateful session beans** For a list of WebLogic features that increase scalability and reliability of stateful session beans, see description in the previous row. **Entity beans** Weblogic Server supports these caching features for entity beans. • Short term or cross-transaction caching • Relationship caching • Combined caching allows multiple entity beans that are part of the same J2EE application to share a single runtime cache Consistency between the cache and the external data store can be increased by: • flushing the cache • refreshing cache after updates to the external data store • invalidating the cache • concurrency control **"read-mostly pattern"** WebLogic Server supports the "read-mostly pattern" by combining read-only and read-write EJBs. **JSPs** WebLogic Server provides custom JSP tags to support caching at fragment or page level.

BETA

Service	J2EE Support	WebLogic Server Scalability and Reliability Features for...
Singleton Services	J2EE APIs used to implement singleton services include: • JMS Destinations, • JTA transaction managers • Cached entity beans with pessimistic concurrency control Scalability can be increased by "partitioning" the service into multiple instances, each of which handles a different slice of the backing data and its associated requests	WLS features for increasing the availability of singleton services include: • Support for multiple thread pools for servers, to harden individual servers against failures • Health monitoring and lifecycle APIs to support detection restart of failed and ailing servers • Ability to upgrade software without interrupting services • Ability to migrate JMS servers and JTA transaction recovery services.

BETA

Application Deployment Considerations

Deploy clusterable objects to the cluster, rather than to individual Managed Servers in the cluster. For information and recommendations, see:

- "Introduction to Two-Phase Deployment" on page 3-5 and
- "Guidelines for Deploying to a Cluster" on page 3-6

Architecture Considerations

For information about alternative cluster architectures, load balancing options, and security options, see "Cluster Architectures" on page 6-1.

Avoiding Problems

The following sections present considerations to keep in mind when planning and configuring a cluster.

Naming Considerations

For guidelines for how to name and address server instances in cluster, see "Identify Names and Addresses" on page 7-4.

Administration Server Considerations

To start up WebLogic Server instances that participate in a cluster, each Managed Server must be able to connect to the Administration Server that manages configuration information for the domain that contains the cluster. 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 contains configuration data for all clustered and non-clustered servers in the Administration Server's domain. You *do not* create a separate configuration file for each server in the cluster.

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 not participate in a cluster. 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 Administration Server.

For these reasons, make sure that the Administration Server's IP address is not included in the cluster-wide DNS name.

Firewall Considerations

If your configuration includes a firewall, locate your proxy server or load-balancer in your DMZ, and the cluster, both Web and EJB containers, behind the firewall. Web containers in DMZ are not recommended. See "Basic Firewall for Proxy Architectures" on page 6-16.

BETA

If you place a firewall between the servlet cluster and object cluster in a multi-tier architecture, 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.

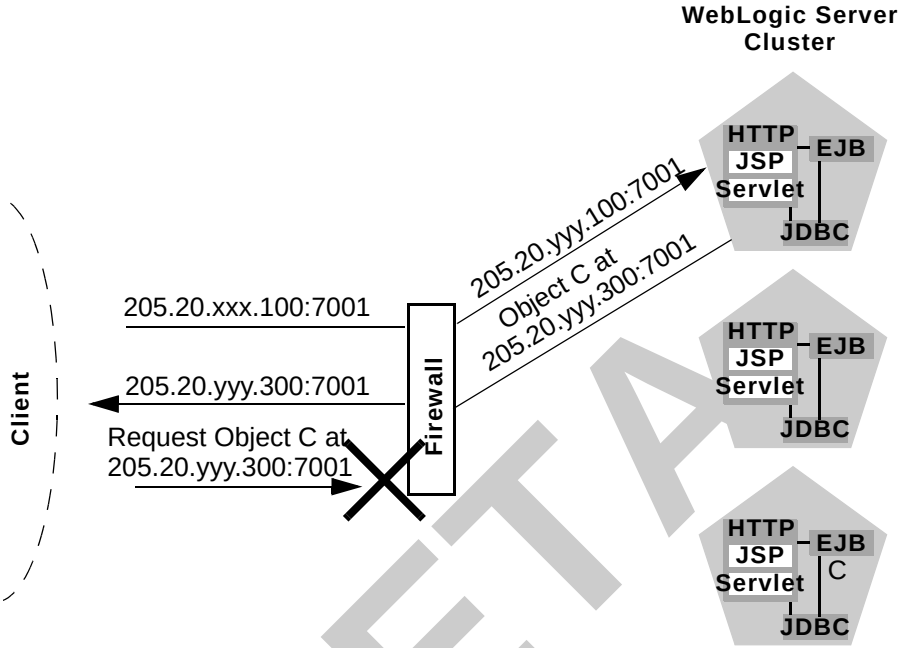
In any cluster architecture that utilizes one or more firewalls, it is critical to identify all WebLogic Server instances using publicly-available DNS names, rather than IP addresses. Using DNS names avoids problems associated with address translation policies used to mask internal IP addresses from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

Notes: Use of `ExternalDNSName` is required for configurations in which a firewall is performing Network Address Translation, unless clients are accessing WebLogic Server using t3 and the default channel. For instance, `ExternalDNSName` is required for configurations in which a firewall is performing Network Address Translation, and clients are accessing WebLogic Server using HTTP via a proxy plug-in.

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.”

BETA

Figure 8-1 Translation Errors Can Occur When Servers are Identified by IP Addresses



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.

BETA

Evaluate Cluster Capacity Prior to Production Use

The architecture of your cluster will influence the capacity of your system. Before deploying applications for production use, evaluate performance to determine if and where you may need to add servers or server hardware to support real-world client loads. Testing software such as LoadRunner from Mercury Interactive allows you to simulate heavy client usage.

BETA

CHAPTER 9

Troubleshooting Common Problems

BETA

This chapter provides guidelines on how to prevent cluster problems or troubleshoot them if they do occur.

Before You Start the Cluster

You can do a number of things to help prevent problems before you boot the cluster.

Check for a Cluster License

Your WebLogic Server license must include the clustering feature. If you try to start a cluster without a clustering license, you will see the error message **Unable to find a license for clustering**.

Check the Server Version Numbers

All servers in the cluster must have the same major version number, but can have different minor version numbers and service packs.

The cluster's Administration Server is typically not configured as a cluster member, but it should run the same major version of WebLogic Server used on the managed servers.

Check the Multicast Address

A problem with the multicast address is one of the most common reasons a cluster does not start or a server fails to join a cluster.

A multicast address is required for each cluster. The multicast address can be an IP number between 224.0.0.0 and 239.255.255.255, or a host name with an IP address within that range.

You can check a cluster's multicast address and port on its Configuration-->Multicast tab in the Administration Console.

For each cluster on a network, the combination of multicast address and port must be unique. If two clusters on a network use the same multicast address, they should use different ports. If the clusters use different multicast addresses, they can use the same port or accept the default port, 7001.

Before booting the cluster, make sure the cluster's multicast address and port are correct and do not conflict with the multicast address and port of any other clusters on the network.

The errors you are most likely to see if the multicast address is bad are:

```
Unable to create a multicast socket for clustering
Multicast socket send error
Multicast socket receive error
```

Check the CLASSPATH Value

Make sure the value of `CLASSPATH` is the same on all managed servers in the cluster. `CLASSPATH` is set by the `setEnv` script, which you run before you run `startManagedWebLogic` to start the managed servers.

By default, `setEnv` sets this value for `CLASSPATH` (as represented on Windows systems):

```
set WL_HOME=C:\bea\weblogic700
set JAVA_HOME=C:\bea\jdk131
.
.
set CLASSPATH=%JAVA_HOME%\lib\tools.jar;
        %WL_HOME%\server\lib\weblogic_sp.jar;
        %WL_HOME%\server\lib\weblogic.jar;
        %CLASSPATH%
```

If you change the value of `CLASSPATH` on one managed server, or change how `setEnv` sets `CLASSPATH`, you must change it on all managed servers in the cluster.

BETA

Check the Thread Count

Each server instance in the cluster has a default execute queue, configured with a fixed number of execute threads. To view the thread count for the default execute queue, choose the Configure Execute Queue command on the Advanced Options portion of the Configuration> General tab for the server. The default thread count for the default queue is 15, and the minimum value is 5. If the value of Thread Count is below 5, change it to a higher value so that the Managed Server does not hang on startup.

After You Start the Cluster

Check Your Commands

If the cluster fails to start, or a server fails to join the cluster, the first step is to check any commands you have entered, such as `startManagedWebLogic` or a `java` interpreter command, for errors and misspellings.

Generate a Log File

Before contacting BEA Technical Support for help with cluster-related problems, collect diagnostic information. The most important information is a log file with multiple thread dumps from a Managed Server. The log file is especially important for diagnosing cluster freezes and deadlocks.

Remember: *a log file that contains multiple thread dumps is a prerequisite for diagnosing your problem.*

1. Stop the server.
2. Remove or back up any log files you currently have. You should create a new log file each time you boot a server, rather than appending to an existing log file.
3. Start the server with this command, which turns on verbose garbage collection and redirects both the standard error and standard output to a log file:

```
% 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
 weblogic.Server >> logfile.txt
```

BETA

Redirecting *both* standard error and standard output places thread dump information in the proper context with server informational and error messages and provides a more useful log.

4. Continue running the cluster until you have reproduced the problem.
5. If a server hangs, use `kill -3` or `<Ctrl>-<Break>` to create the necessary thread dumps to diagnose your problem. Make sure to do this several times on each server, spaced about 5-10 seconds apart, to help diagnose deadlocks.

Note: If you are running the JRockit JVM under Linux, see "Getting a JRockit Thread Dump Under Linux" on page 9-4.

6. Compress the log file using a Unix utility:

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

   - or zip it using a Windows utility.

7. *Attach* the compressed log file to an e-mail to your BEA Technical Support representative. Do not cut and paste the log file into the body of an e-mail.
8. If the compressed log file is too large, you can use the BEA Customer Support FTP site.

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 Command Reference*.
- 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.

BETA

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.

Check Garbage Collection

If you are experiencing cluster problems, you should also check the garbage collection on the managed servers. If garbage collection is taking too long, the servers will not be able to make the frequent heartbeat signals that tell the other cluster members they are running and available.

If garbage collection (either first or second generation) is taking 10 or more seconds, you need to tune heap allocation (the `msmx` parameter) on your system.

Run utils.MulticastTest

You can verify that multicast is working by running `utils.MulticastTest` from one of the managed servers. See "Using the WebLogic Server Java Utilities" in *WebLogic Server Command Reference*.

BETA

BETA

The WebLogic Cluster API

The following sections describe the WebLogic Cluster 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.

BETA

Each server in the cluster is uniquely identified by its name as defined with the WebLogic Server 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;
    }
  }
}
```

BETA

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 4-12.

BETA

BETA

Configuring BIG-IP™ Hardware with Clusters

BETA

This section describes options for configuring an F5 BIG-IP controller to operate with a WebLogic Server cluster. For detailed setup and administration instructions, refer to your F5 product documentation.

- Configuring Session Persistence
- Configuring URL Rewriting

For information about how WebLogic Server works with external load balancers, see "Load Balancing HTTP Sessions with an External Load Balancer" on page 4-2.

Configuring Session Persistence

BIG-IP supports multiple types of cookie persistence. To work with a WebLogic Cluster, you must configure BIG-IP for the Insert Mode form of HTTP Cookie Persistence. Insert mode insures that the 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.

Configuring URL Rewriting

BIG-IP Version 4.5 provides support for URL rewriting.

Configuring WebLogic Server for URL Rewriting

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. In addition, 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. For instructions on how to enable URL rewriting, see "Using URL Rewriting Instead of Cookies", in *Developing Web Applications for WebLogic Server*.

Configuring BIG-IP for URL Rewriting

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.

BETA

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

BETA

BETA