Oracle9*i* Application Server

Best Practices

Release 1 (v1.0.2.2)

September 2001

Part No. A95201-01

ORACLE

Oracle9*i*AS Best Practices, Release 1 (v1.0.2.2)

Part No. A95201-01

Primary Authors: Craig B. Foch, Alice Chan, Matthieu Devin, Gary Hallmark, and Bruce Lowenthal

Contributors: Dan Damon

Contents

Send Us Your Comments ix

Preface xi

Audience xii
Structure xii
Related Documentation xiii
Conventions xiv
Documentation Accessibility xx

1 Availability of Java Applications

Availabilty Overview 1-2
Key Practices 1-2
Measuring Availability 1-3
Hardware Availability 1-4
Software Availability 1-5
Data Availability and Reliability of Operational Procedures 1-6
Users' View of Application Availability 1-6
Hardware Redundancy and Load Balancers 1-7
Firewalls 1-9
Clustering 1-10
Application Server Cluster 1-11
Web Cache Cluster 1-12
Firewall Clusters 1-12
Commercial Balancers 1-13

HA-HW-1: Hardware Components 1-14
HA-HW-2: Load Balancers 1-14
HA-HW-3: Configure mod_jserv 1-15
HA-HW-4: Independent Servers 1-16
Software Design Principles 1-17
HA-SW-1: Java Synchronization 1-17
HA-SW-2: Resource Use 1-18
HA-SW-3: Session State 1-19
HA-SW-4: Catch-All Exceptions 1-21
HA-SW-5: Finally Clause 1-22
HA-SW-6: Retry Transactions Once 1-22
HA-SW-7: Database Update 1-23
Periodic Operations Procedures 1-24
HA-OP-1: Backup Schedule 1-24
HA-OP-2: Server Restarts 1-25
HA-OP-3: Mid-Tier File Synchronization 1-27
HA-OP-4: Document Procedures 1-28

2 Performance and Scalability

Performance and Scalability Overview 2-2
Performance Tradeoffs 2-2
Servlets and JavaServer Pages: A Case Study 2-3
User Information 2-4
Use of Hashtables 2-5
endRequest() Called Manually 2-5
Servlets and JavaServer Pages: Design and Implementation 2-6
PERF-1: Storing State Information 2-6
PERF-2: Connection Pooling and Caching 2-9
PERF-3: Statement Caching 2-10
PERF-4: Mid-Tier Caching 2-11
PERF-5: Database Connections 2-11
PERF-6: Database and SQL Tuning 2-12
PERF-7: Memory Leaks 2-12
PERF-8: Spawning Threads 2-13
PERF-9: SingleThreadModel in Servlets 2-13

PERF-10: Servlets Startup Operations 2-15
PERF-11: Separate JSPs 2-15
PERF-12: Static or Dynamic Include in JSPs 2-15
PERF-13: Thread-Safe JSPs 2-16
PERF-14: Sessions in JSPs 2-17
PERF-15: Use Forward in JSPs 2-17
PERF-16: JSP Buffer 2-18
Servlets and JavaServer Pages: Deployment 2-18
PERF-17: Multiple JServs 2-19
PERF-18: Java Heap Size 2-19
PERF-19: JServ autoreload.classes 2-20
PERF-20: Preload Servlets 2-20
PERF-21: Pre-translate JSPs 2-20
PERF-22: JSP debug flag and developer_mode 2-21
PERF-23: Dynamic Monitoring Service 2-21
Java Performance 2-21
PERF-24: Synchronization 2-21
PERF-25: Hashtable and Vector 2-25
PERF-26: Reuse Objects 2-26
PERF-27: Unused Objects and Operations 2-28
PERF-28: StringBuffer 2-28
Oracle HTTP Server Tuning 2-29
PERF-29: TCP/IP Parameters 2-30
PERF-30: KeepAlive 2-30
PERF-31: MaxClients 2-30
PERF-32: DNS Lookup 2-31
PERF-33: Access Logging 2-31
PERF-34: SSLSessionCacheTimeout 2-31
PERF-35: FollowSymLinks 2-32
PERF-36: AllowOverride 2-32
PERF-37: DMS and Apache Server Status 2-32
Performance Testing 2-32
PERF-38: Performance Methodology 2-33
PERF-39: Performance Goals 2-33
PERF-40: Performance Evaluation 2-33

PERF-41: Performance Testing 2-33
PERF-42: Test Driver 2-34
PERF-43: Testing Personnel 2-34
Dynamic Monitoring Service (DMS) 2-34
Code Example 2-37
References and Resources 2-43
Oracle Documentation 2-44

3 Pooling

Pooling Overview 3-2
Pooling Example 3-3
Pooling versus Sharing 3-3
Pools in Oracle9*i*AS 3-4
Apache Demons 3-4
POOL-1: Client-Request Processes 3-4
JDBC Connections 3-5
POOL-2: JDBC Connection Cache 3-5
POOL-3: Connection Authentication 3-5
Servlets and Connection Cache 3-6
POOL-4: Database Updates 3-8
POOL-5: Linked Servlets 3-9
Holding Connections Across Calls 3-10
POOL-6: min and max Parameters 3-11
POOL-7: Cache Usage 3-12
Multiple Applications in the Same Java VM 3-13
Note for Multithreaded Servlets 3-13
POOL-8: SQLJ 3-14
POOL-9: JDBC Statement Cache 3-15
JDBC OCI Connection Pools 3-15
BC4J Application Modules 3-16
POOL-10: Application Module Pools 3-16
Servlets and Jserv threads 3-18
POOL-11: Servlet Attributes 3-18
JavaServer Pages (JSPs) 3-20
POOL-12: ConnBeans 3-20

POOL-13: Single-Threaded JSPs 3-21
Perl 3-22
POOL-14: Perl Programs 3-22
Portal or PL/SQL 3-22
Database Shared Servers 3-22
POOL-15: Shared Server Mode 3-23
Pooling Your Own Resources 3-23

4 HTTP Security

Security Overview 4-2
Firewall Architecture 4-2
SEC-1: Server Placement 4-5
SEC-2: Restrict Traffic Through Exterior Firewall 4-5
SEC-3: Restrict Traffic Through Interior Firewall 4-6
SEC-4: Proxies on the DMZ 4-6
SEC-5: Safeguard Information of Record 4-6
SEC-6: Restrict Traffic to Approved Types 4-6
Process Development and Deployment 4-8
SEC-7: Privileges and Modules 4-8
SEC-8: Buffer Overflow 4-8
SEC-9: Cross-Site Scripting Attacks 4-9
SEC-10: Exported Products 4-9
SEC-11: Passwords and Accounts 4-10
SEC-12: Security Patches 4-10
SEC-13: Unused Services 4-10
SEC-14: Root Privileges 4-10
SEC-15: "r" Commands 4-10
Global Server ID Certificates and 128-Bit Encryption 4-11
SEC-16: Fail Weak Encryption 4-13
Performance Issues 4-14
SEC-17: Performance Testing 4-15
SEC-18: Sequential HTTPS Transfers 4-15
SEC-19: Separate Virtual Servers 4-15
Client Certificates 4-16
SEC-20: Organization Identity 4-18

SEC-21: User Identity 4-18
SEC-22: Expiring Certificates 4-18
SEC-23: Certificate Reissues 4-19
SEC-24: Certificate Revocations 4-19
EXPORT versus DOMESTIC Encryption Issues 4-20
Export Law Changes 4-20

5 Using Oracle9*i*AS Web Cache with Third-Party Servers

Oracle9*i*AS Web Cache Overview 5-2
Integration with Third-Party Application Servers 5-2
Web-site Configuration 5-3
Cacheability Rules and Expiration Rules 5-5
BEA WebLogic 6.0 Java Applications 5-5
WebLogic SnoopServlet 5-5
WebLogic SessionServlet 5-7
IBM WebSphere 3.5.2 Java Applications 5-10
WebSphere Snoop Servlet 5-10
WebSphere SessionSample 5-11
Microsoft IIS 5.0 ASP Applications 5-14
ServerVariables_Jscript ASP 5-14
Cookie_Jscript ASP 5-15

Glossary

Index

Send Us Your Comments

Oracle9*i*AS Best Practices, Release 1 (v1.0.2.2)

Part No. A95201-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail - iasdocs_us@oracle.com
- Fax - (650) 506-7409 Attn: Oracle9*i* Application Server Documentation Manager
- Postal service:

  Oracle Corporation
  Oracle9*i* Application Server Documentation Manager
  500 Oracle Parkway, M/S 2op4
  Redwood Shores, CA 94065 USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This manual presents best practices for Oracle9*i* Application Server Release 1 (v1.0.2.2).

This preface contains these topics:

- Audience
- Structure
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

This manual is intended for two groups of users. One group consists of software architects, consultants, and developers who:

- Build applications on Oracle9*i* Application Server
- Build applications on Oracle HTTP Server

The other group of users consists of IT managers, purchasers, and staff who:

- Deploy and manage applications on Oracle9*i* Application Server
- Deploy and manage applications on Oracle HTTP Server
- Run highly available Web sites

To use this document, you need familiarity with Java, servlets, JavaServer Pages, and Apache.

Structure

This manual contains:

Chapter 1, "Availability of Java Applications"

This chapter presents Oracle recommendations for keeping Web-based information systems operating around the clock, day in and day out—commonly known as high availability.

Chapter 2, "Performance and Scalability"

This chapter provides a checklist of what we consider performance and scalability best practices in designing, implementing, tuning, and testing your Web application that is to be deployed on Oracle9*i*AS Release 1 (v1.0.2.2).

Chapter 3, "Pooling"

This chapter discusses pooling mechanisms, which increase application scalability by allowing many users to access the same application without an exorbitant tax on server resources.

Chapter 4, "HTTP Security"

This chapter provides an overview of HTTP security as well as detailed recommendations regarding firewalls, process development and deployment, certificates, and encryption issues.

Chapter 5, "Using Oracle9iAS Web Cache with Third-Party Servers"

This chapter first discusses the integration of Oracle9*i*AS Web Cache with third-party application servers from a general perspective, followed by detailed examples involving three particular application servers.

Glossary

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Application Server Overview Guide* in the Oracle9*i* Application Server Documentation Library
- *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation
- *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library
- `http://metalink.oracle.com/`
- `http://technet.oracle.com/index.html`

In North America, printed documentation is available for sale in the Oracle Store at

- `http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

- `http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

- `http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

- `http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

- http://tahiti.oracle.com

For additional information, see:

- http://csrc.nist.gov/publications/fips/fips1401.htm
- http://cwis.kub.nl/~frw/people/koops/lawsurvy.htm
- http://isglabs.rainbow.com/isglabs/shawn/SSL_Perf/otpssl8.html
- http://www.apache.org/info/css-security/
- http://www.cert.org
- http://www.checkpoint.com/
- http://www.iso.ch
- http://www.verisign.com/

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Microsoft Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

Convention	Meaning	Example

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Conventions for Microsoft Windows Operating Systems

The following table describes conventions for Microsoft Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Oracle Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"`
		`C:\>imp SYSTEM/`*`password`*
		`FROMUSER=scott TABLES=(emp, dept)`

HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin95` for Windows 95 ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install Oracle9*i* release 1 (9.0.1) on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora90`. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

1

Availability of Java Applications

This chapter presents Oracle recommendations for keeping Web-based information systems operating around the clock, day in and day out—commonly known as high availability. Our recommendations take the form of best practices for:

- Programming highly available Java server applications
- Deploying them to a redundant hardware environment
- Operating them proactively with frequent restart and backup

We recommend a combination of:

- Backup of operating system and database files
- Hardware failure detection and routing using **load balancers**
- Periodic server process reboots to reduce software failures

Failure detection is effective for total failures like the failure to accept a TCP connection. Detection is much less effective for partial failures, like a process that accepts the connection but does not reply within a few seconds. These partial failures are often caused by deadlocks, resource leaks, and other server-side software bugs. Software failures often cause *death by a thousand cuts*. At no single event or time can a failure detector say *it failed*. For some slow software failures, it is much easier to simply kill and restart processes or computers periodically, before they get into trouble.

This chapter contains these topics:

- Availabilty Overview
- Hardware Redundancy and Load Balancers
- Software Design Principles
- Periodic Operations Procedures

Availabilty Overview

To keep Web-based information systems operating around the clock day in and day out, robust software must be carefully executed on redundant hardware. Overall Web-site availability requires good practices in programming, deployment, and operations. Weakness in one area can compromise overall availability, even if the other areas follow good practices.

This section contains these topics:

- Key Practices
- Measuring Availability
- Hardware Availability
- Software Availability
- Data Availability and Reliability of Operational Procedures
- Users' View of Application Availability

Key Practices

Key deployment practices include:

- Analyze your network topology for single points of failure.

  If any component is not much more reliable than the targeted overall system reliability, then consider installing redundant components. Topology elements include routers, switches, firewalls, load balancers, cables, and power supplies.

- Use more than one mid-tier machine, fronted by a load balancer that balances loads over available machines, skipping failed and out-of-service machines.

  Examples of load balancers are Cisco's CSS 11000 and F5's BigIP.

Key programming practices include:

- Store per-client state in the database tier between requests.
- Use monitoring tools to track use of Java threads, synchronization, and shared resources to quickly diagnose deadlocks and resource leaks.
- Do not reuse objects that throw potentially serious errors, so that retries are more likely to work.
- Retry at a coarse granularity, such as entire HTTP requests or entire transactions.

Key operational practices include:

- Proactively restart JVMs daily rather than waiting for them to crash.
- Synchronize backups of operating system files with database contents so that you can restore your entire Web site from scratch in a reasonable time.

  Examples of such files are static HTML content, Java code, XML descriptors, and configuration files.

These mid-tier practices put much of the burden for overall Web-site availability on the database; therefore it is important to understand how to make the database highly available. Database availability is not covered in depth in this manual.

> **See Also:** For more information on database availability, see the *Oracle9i Database Administrator's Guide* or *Oracle8i Database Administrator's Guide* in the Oracle Database Documentation Library

Measuring Availability

Availability of an overall system or of an individual system component is defined as the percentage of time that it meets correctness and performance specifications. A component that meets specifications 8 hours a day has 33.3% availability, as does a component that works 20 minutes every hour. A system or component that has 99% availability is, on average, down 3.65 days per year.

Mission-critical systems often have goals of 4 nines (99.99% availability), which is less than an hour downtime per year, or even 5 nines (99.999% availability), which is 5 minutes of downtime per year.

Availability may not be constant over time. It may be higher during the day shift, for example, when most transactions are expected to take place, and lower during the night shift, when planned downtime may be needed to install hardware or software upgrades. Because the Internet is global in reach, however, there is less flexibility to time-shift availability requirements. It is common to require availability every hour of every week, often referred to as 24x7.

Redundant components can be used to improve availability, but only if they can take over for the failed primary quickly. For example, if a light bulb burns out after an hour of use, but it takes 10 minutes to detect the darkness and 20 minutes to replace the bulb, then the availability of light is 60/90 = 66.6%.

Planning for a highly available Web site requires first establishing an overall availability goal and then taking a critical look at the cost and availability of every piece of hardware, every module of software, every byte of data, and every task in the day-to-day operations to determine where an increase in component availability increases overall availability with reasonable cost.

Hardware Availability

Hardware is probably the easiest to analyze, because component availability may be specified by the manufacturer, and components can usually be assumed to fail independently. Independence allows us to calculate the availability of a system consisting of two components C1 and C2 that must both be available as

Av(C1 and C2) = Av(C1) * Av(C2)

The availability of a system of two redundant components, only one of which must be available, assuming zero time to detect a failure in one component and switch over to the spare, is

Av(C1 or C2) = 1-(1-Av(C1))(1-Av(C2))

In general, as more non-redundant components are added, availability decreases, and as more redundant components are added, availability increases. A *Noah's Ark* approach to hardware availability is to simply have two of everything. This is adequate when components have similar availability and cost. However, as the following example shows, more detailed analysis can often save money.

The goal in this example is to achieve Av(C1 and C2) > .99. Assume that:

- Av(C1) = .97
- Av(C2) = .992
- C1 costs $100
- C2 costs $1000

Using the formulas above, we get

- Av(C1 and C2) = .97 * .992 = . 962

  Without any redundancy, availability is below the goal.

- Av((C1 or C1) and (C2 or C2)) = (1-.0009)(1-.000064) = .999

  Noah's Ark redundancy (two of everything) exceeds the goal, but costs $2200.

- Av((C1 or C1) and C2) = (1-.0009)*.992 = .991

  Replicating just C1 meets the goal and costs only $1200.

If the redundant components can dynamically check each other and failover automatically (rather than require manual intervention), then recovery will be quicker. It is even better if the extra pieces of hardware can work simultaneously to increase throughput or decrease response time in the case of no failure, although adequate capacity should exist when a failure does occur.

Modern hardware typically fails very infrequently (only after many months or even years), so the primary factors affecting hardware availability are:

- Not having a spare on hand
- Not being able to diagnose what component has failed
- Not being able to swap out the failed part without causing other components to fail or be shut down

Software Availability

Software can fail because of underlying hardware failure, but software usually fails because of logic defects (that is, bugs). All software that is complex enough to provide interesting Web-based applications has bugs. It is not unusual for complex Java server applications to crash daily under heavy load. Such server processes can take several minutes to restart, yielding an availability of about 99.5%.

Java bugs do not always immediately crash their containing process. Memory leaks can make the process sluggish, so that it is only partially available. Because of the inherent difficulty in detecting software failures, and the lack of good software failure detectors for JVMs in the current Oracle9*i*AS product, our recommended best practices are:

- Follow good software design practices, so that Java servers can stay up for at least a day under heavy load.
- Adopt a proactive operational practice of restarting JVMs daily (approximately) to prevent availability of Java servers from dropping below 99% due to bug-induced partial availability.
- Spread requests over multiple JVMs, so that overall JVM availability can be made much higher than 99.5%.

Data Availability and Reliability of Operational Procedures

Data can be lost or corrupted due to hardware failures, software failures, intentional acts (viruses or hackers) and unintentional acts (operations staff runs the wrong batch job, for example, or backup data sets are mislabeled or lost).

To prevent data loss, solid file system and database backup practices are essential, as are training and fire drills for operations staff. To improve data integrity, use database transactions and integrity constraints. To prevent modification to static Web pages and Java code, deploy them on read-only file systems or use intrusion-detection software.

To minimize operational mistakes, good system administration tools are important. These include graphical displays of topology and memory, CPU, and network activity. Good training is also important. Be wary of a hodgepodge of homegrown shell scripts without robust error checking and data validation logic.

Operations staff must record failures, and these failures must be analyzed periodically. Hardware failures that occur more often than the manufacturer's specifications indicate either adverse machine room conditions or the need for a different manufacturer. Software failure records aid in bug tracking, justifying more code profiling and monitoring tools, or refining the period (for example, daily or weekly) for restarting JVMs or Web servers. Operational failure records often suggest needed improvements in tools, scripts, or training.

Users' View of Application Availability

In a perfect world, users would benefit transparently from all the work that system administrators and programmers put into a highly available Web site. All application screens would appear in less than a second or two, *server not found* or *internal server error* screens would never appear, users would never need to repost a form after receiving one of these errors, and you would never need to search through application logs to find out if the first post succeeded or failed. Unfortunately, the nature of HTTP and commercial Web browsers makes it impossible to mask all failures.

Even in an imperfect world, however, users should never be exposed to corrupt, missing, or incomplete high-value data such as payment transactions, addresses, or phone numbers. To allow for flexible implementation, low-value data such as catalog information or news feeds are often allowed to be less consistent.

When application data have been recovered to a consistent state, users should always be able to start a new application session. A large class of service failures can be handled by invalidating existing sessions and requiring the user to restart his

session. While this procedure is not transparent to users, they can usually recover and resume their work.

When in session, a failure to process a single request should not cause the session to fail. Session failure may be seen by a user if multiple requests fail or if the session expires due to inactivity or administrative intervention.

Requests may fail with no indication as to whether session or transactional data was modified. Users may resubmit requests, whether failed or otherwise, using reload or refresh browser functions without starting a new session, but in general it is the user's responsibility to ensure that the request is **idempotent**. This is the typical behavior for e-commerce and e-brokerage sites. Indiscriminate retry after a failure can result in duplicate transactions, although applications typically use logs to warn of potential problems.

Many failures of idempotent and transactional requests can be retried transparently. Failures that occur in the scope of a servlet's doGet or doPost method can be transparently retried by try..catch logic. Good application software design will relieve users of retry responsibility, except for failures that occur between the browser and the servlet engine.

Hardware Redundancy and Load Balancers

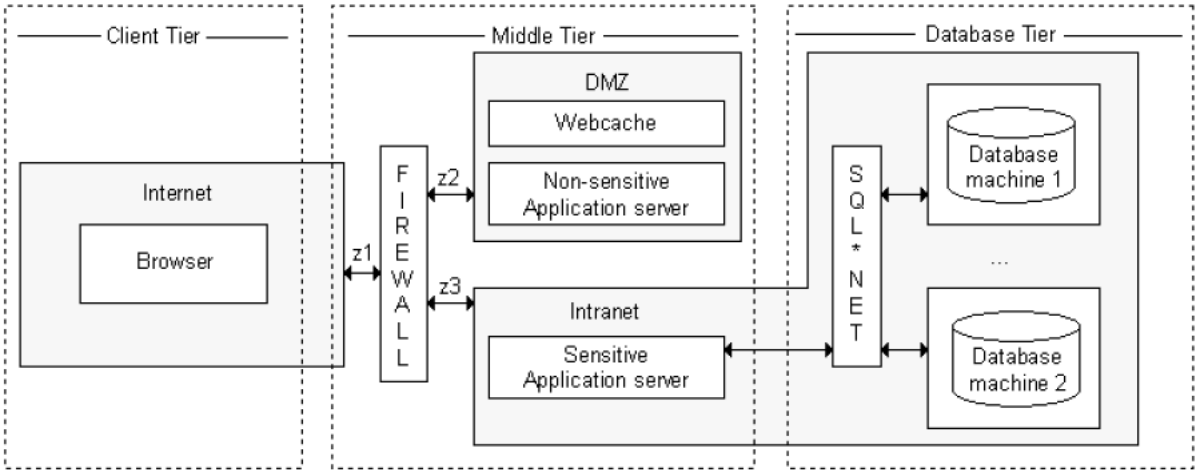
All high-availability solutions have the same general approach. Keep redundant components on hand to use when something breaks. When something does break, switch automatically and quickly.

In the reference topology illustrated in Figure 1–1, any of the following components can break:

- Application server computers running Apache, JVM processes, and other Oracle9*i*AS processes
- Web cache computers
- Firewall computers
- Database computers
- Data storage devices
- Network file system servers
- Network appliances such as switches, routers, and load balancers
- Network cables

- Power supplies
- Internet service provider hardware in the Internet

Figure 1–1 Reference Topology



In this chapter, we are primarily concerned with availability of mid-tier components. It is equally important to have a thorough understanding of database availability issues and Internet service provider (ISP) availability issues, including redundant ISPs and the Boundary Gateway Protocol for ISP-facing routers.

Application server computers, Web cache computers, and firewall computers can be clustered using hardware load balancers to increase overall availability and provide scalability. Each computer in the cluster is independent but identically configured. The identical configuration is maintained using scripts to copy the same contents to each computer's local disk, or a highly available shared disk system is used.

Load balancers are balancing, failure detecting, multi-layer (Open Systems Interconnect network reference model layers 2-7) switches like Cisco's CSS 11000 or F5's BigIP, which can detect failure of clustered mid-tier servers and route around them. Load balancers also allow mid-tier servers to be taken out of service gracefully. Once out of service, a mid-tier server may be reconfigured or simply rebooted in order to reduce the chance of a crash or slowdown due to resource leaks and other bugs.

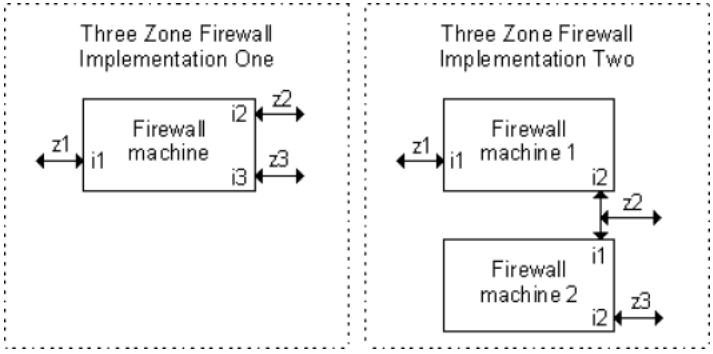
Network appliances should all support a primary/backup configuration. Cabling and power supplies need special attention and are often the source of problems. For example, there are stories of a single slice of a carpenter's saw severing both primary and backup networking cables.

In general, redundant components should have independent failure modes. Clustered computers should not have a single power supply, single network cable, or single shared disk or file server. (Exceptions can be made for single components with very high availability.) Backups of your database and operating system files should not be kept in the computer room, where they might burn up along with the online copy.

Firewalls

Firewalls separate the global network into three security zones. The two implementations shown in Figure 1–2 are common.

Figure 1–2 Firewall Implementations



Implementation One shows the simplest way to implement a firewall with three zones. A single computer with three network interface cards (i1, i2, and i3) running appropriate software divides the network into zones z1, z2, and z3. Cisco's PIX, Checkpoint's FW-1, Symantec's Raptor, and many others can work this way. All these firewalls optionally support a second standby firewall machine that can take over should the primary fail.

Implementation Two is an alternative that uses two machines, each with two network interfaces, to achieve three zones. This approach may be more difficult to manage, because there are two sets of security rules that are not centralized. It is

most useful when the duties of firewall machine 1 can be assumed by a router, load balancer, or other already-existing machine. It is also more challenging to configure Implementation Two for failover, because you must configure both machines for failover and then test to make sure the combination will failover in case machine 1 or machine 2 fails.

Instead of employing a passive standby firewall for availability, both firewall implementations may be clustered with load balancers to achieve both availability and scalability.

Clustering

In this chapter, a mid-tier cluster is defined as a set of processes (servers) which can all provide more or less the same service. Usually, the servers are spread over several computers in order to make failures more independent, as well as to provide scalability. Requests to servers can be routed with hardware load balancers or software clustering services. Software solutions have several drawbacks:

- Reliance on DNS round robin to map a single domain name to multiple IP addresses

  This DNS trick cannot respond to dynamic load changes or failures.
- Long response times during which many requests fail
- Not implemented for all Oracle9*i*AS components
- Risk making the clustered processes interdependent rather than independent

Hardware load balancers offer several advantages:

- Short response times
- Support for taking individual servers offline for maintenance without a lot of cluster synchronization overhead
- Sticky routing of requests to servers based on sessions, URLs, and network addresses, which allows effective clustering of stateful servers, whether the state is SSL or HTTP sessions, cache contents, or **stateful inspection** firewalls

In Figure 1–3, Figure 1–4, and Figure 1–5, load balancers labeled balancer1, balancer2, balancer3, and balancer4 all provide scalability and failover. Failover can hide both planned and unplanned outages. The balancers in the diagrams are logical.

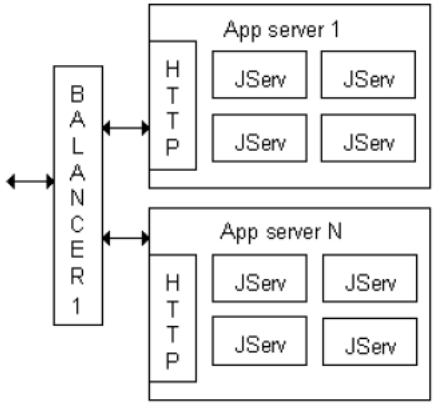
Many logical balancers can be mapped to a single physical load balancer device using load balancer-specific configuration commands. The commands to define a logical balancer typically associate an input address (MAC or IP) with several output addresses, some load balancing options, and some routing options (also called sticky modes). There may be balancer-specific limits on the number and type of logical balancers that can be configured, and there are network throughput limits as well.

In this chapter we assume that the number of clustered servers is in the range 2..50. This is well within range for most applications of hardware load balancers. More importantly, with 50 or fewer clustered servers, you can use single-threaded scripts to cycle through them one at a time to run administrative scripts. Managing larger clusters would require more administrative support specifically targeted at clusters than currently exists in Oracle9*i*AS.

Application Server Cluster

A single application server consists of the Oracle HTTP Server and a couple of JVM processes for each CPU running Jserv, JavaServer Pages (JSPs), and perhaps other Oracle9*i*AS components (although the other components are not covered in this chapter). Application servers can be clustered using load balancers in order to increase availability and throughput.

Figure 1–3 Application Server Cluster



In Figure 1–3, Balancer1 spreads connection requests to a virtual IP address and port over several real servers, based on such policies as round robin, static

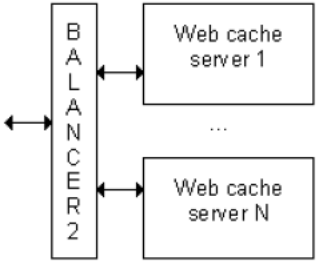
weighting, or least average response time. It supports stickiness based on SSL sessions and HTTP cookies. That is, SSL connections that are part of the same secure session will be routed to the same target machine, saving session key negotiation, and HTTP connections that are part of the same HTTP session will be routed to the same target machine, saving having to migrate session state.

Web Cache Cluster

A single Web cache server caches frequently accessed Web pages in RAM. The cache does not spill to disk, and the cache server process is single-threaded. Web cache servers can be clustered using a load balancer for greater availability, memory capacity, and throughput.

In Figure 1–4, Balancer2 supports stickiness based on URL patterns, which can be used to partition cached Web pages over a cache farm. For example, Balancer2 could be configured to direct requests for URLs whose **hash** is even to server 1, and whose hash is odd to server 2.

Figure 1–4 Web Cache Cluster



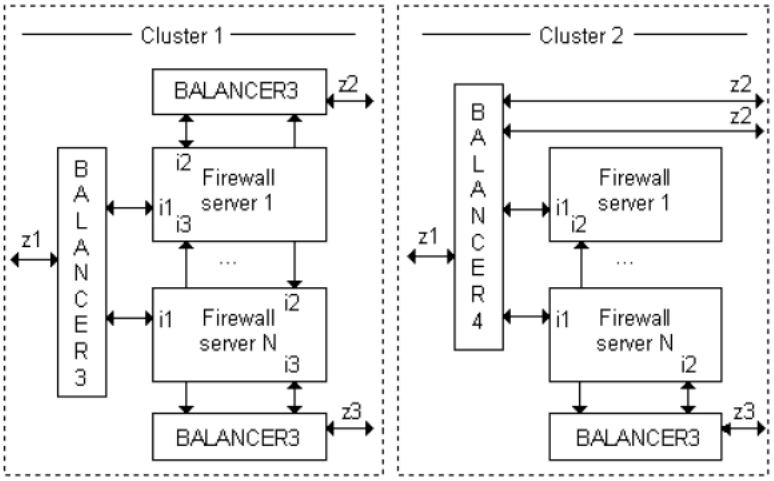
Firewall Clusters

An enterprise firewall solution will usually be made up of several machines for availability and scalability. In Figure 1–5, Balancer3 supports stickiness based on (source IP, destination IP) pairs, which ensures that all traffic for a given inbound or outbound connection can be inspected by the same firewall server. These balancers must be hardened so that they are not victims of denial-of-service attacks themselves. For a firewall cluster of M servers implementing N security zones, N balancers, one in each security zone, should work together to balance traffic over the M firewall servers. The diagram shows N=3 zones: Internet, DMZ, and intranet.

Balancer4 supports balancer2 and balancer3 capability, as well as enough firewall capability to isolate z1 and z3.

In order to guarantee that network packets cannot be routed around firewalls due to misconfiguration, all logical balancers in a security zone must map to physical balancers in that same zone.

Figure 1–5 Three-Zone Firewall Clusters



Commercial Balancers

Commercial load balancers that support all the sticky modes for application as either balancer1, balancer2, balancer3, or balancer4 include:

- Cisco CSS 11000 (acquired via Arrowpoint).

  Note that LocalDirector is older and cannot serve in the role of balancer2, balancer3, or balancer4.

- F5 Systems BigIP
- Foundry Networks ServerIron

HA-HW-1: Hardware Components

Simulate failure and replacement of every hardware component, including redundant spares. Measure the actual availability impact.

For this exercise, adopt a pessimistic, Murphy's Law attitude: if it can break, it will! Power off server machines, routers, load balancers, and firewalls. Unplug network cables. Unplug disk drives.

For each failure, does failover occur automatically? How long does it take a system administrator to locate the failure? Ideally, a management framework will issue specific alerts targeting the failed component until it is repaired. After a real failure, be sure to order replacement parts promptly.

Finally, what is the impact of repair? Are components hot-pluggable so that the repair can be effected without shutting down other components? Repair, reconfiguration, or just adding a new server to a cluster to increase capacity can be major sources of downtime.

Note: If components are not hot-pluggable, then you should be careful not to cause real failures when unplugging components for this exercise.

Having experienced a simulated failure in training can greatly decrease the risk of a bad problem being made worse by inexperienced operations staff under pressure.

HA-HW-2: Load Balancers

Use commercial load balancers to cluster application server computers and Web cache computers to achieve availability and scalability.

Load balancers like Cisco's CSS 11000 and F5's BigIP can quickly detect and route around failed mid-tier servers. For greatest failure isolation, each server should be an independent computer, although if the availability of the computer and its operating system is very high, then multiple servers may run on the same computer. Load balancers can also quiesce and take a mid-tier server offline, so that it can be restarted or reconfigured without having to handle incoming requests at the same time.

Ideally, N+2 mid-tier machines should be used, where N is the number of mid-tier computers needed to serve peak load. One extra computer is a ready-to-go spare, should one of the N fail. The other extra computer hosts a server that is offline for restart or maintenance.

Load balancers have failover configurations to prevent themselves from becoming single points of failure.

Load balancers can also be used to balance traffic across multiple firewall computers for scalability and availability. This can be a complex and expensive option, because you must have a load balancer in each security zone. Most commercial firewalls have a failover solution that does not require use of a load balancer. So unless scalability is a concern, use the availability solution recommended by the firewall vendor.

HA-HW-3: Configure mod_jserv

Each clustered application server runs one instance of Apache for each server and two Jserv processes for each CPU. Module mod_jserv should route requests within a single application server, not across the cluster.

Multiple processes on a single computer can help isolate software failures. You want enough Jserv processes so that some can fail and enough survive to fully utilize the computer resources until the next JVM restart cycle (see Chapter 2, "Performance and Scalability").

Module mod_jserv (using the Apache Jserv Protocol, or AJP) can be configured to route servlet requests from the Apache HTTPD processes to both local Jserv processes and to remote Jserv processes in different application servers in the cluster. It is best to configure mod_jserv to route only to local Jserv processes because:

- The AJP protocol, vulnerable to sniffing attacks, will not appear on any LAN.
- It is much easier to quiesce an application server and all its associated JVMs.

  Simply quiesce the HTTP requests to the Apache instance. This makes it possible to take an application server instance out of the cluster for maintenance without aborting in-flight user requests.
- Load balancing in mod_jserv and in the load balancer itself will not conflict.

Trade-offs with this practice include:

- Apache processes and Jserv processes cannot be separated by a firewall.

  But firewalls may not statefully inspect protocols such as AJP. A more secure solution is to route HTTP through the firewall from Web cache or other HTTP proxy in a less secure zone to an application server cluster in a more secure zone.

- If SSL or HTTP sessions are used, then the load balancer should be set to use sticky routing so that SSL sessions are not re-negotiated and HTTP session state does not have to be migrated.

HA-HW-4: Independent Servers

Each clustered server should be configured identically but be as independent as possible.

Independence means a mid-tier server can be added to a cluster, removed from a cluster, or fail without affecting the clustered servers. Strictly speaking, if one clustered server fails, then the others are somewhat affected because their loads will increase. But to a good first approximation, only the load balancer needs to be reconfigured when servers are added and removed, and only the load balancer needs to detect and react to server failure automatically.

Resources that are shared by the mid-tier servers must be made highly available themselves. For example, the database, shared NFS server, network hubs, and routers must be made highly available. And if multiple servers are hosted on a single computer, then that computer and its operating system need to be more robust than a computer that is hosting a single server.

For independence, mod_jserv's configuration should connect Apache to the local computer's Jserv processes and not to Jserv processes on other computers. Each mid-tier computer is a *failure container*. It is best for the load balancer to route requests among the failure containers, and for each Apache to route requests solely within a failure container.

To ease management of the multiple computers, it helps to configure them as similarly as possible. Small performance differences can easily be handled with load balancing built into the load balancer. Large performance differences can be handled by running two or more application servers on a single big machine. It may be difficult to take the fastest computers completely offline for maintenance and still have enough capacity left over for peak load.

Software Design Principles

Many of the failures you need to handle are software problems. It is difficult to write efficient server software that handles request after request forever (assuming no hardware failure). Our goal is to write software that can stay up for about a day, and then you can reboot it.

The software design practices in this chapter are targeted to Java server programs such as servlets and JSPs that run in a standard JVM in an operating system process. The primary sources of bugs are:

- Use of concurrent threads and errors synchronizing access to shared data
- Resource leaks and poisoned pools (usually caused by improper error handling and cleanup)

One of the most important software design principles is that all mid-tier applications should be *statesafe*. This term is used instead of **stateless**, because we want to emphasize that it is common and normal for mid-tier application servers and caches to have a lot of state that is shared or serially reused by many requests from many users. But the per-user state (also known as session state) should be stored someplace safe between client requests. Safe places to store session state include browser cookies and a backend database, but not a single JVM process that could crash or be restarted at any time.

The Jserv implementation of HttpSession is not statesafe. That is, it does not secure a copy of its data in either the browser or the database between user requests. Therefore, HttpSession cannot be used to store recoverable session information.

HA-SW-1: Java Synchronization

Carefully monitor the use of Java synchronization. Synchronization can cause a deadlock—a situation when throughput and resource utilization both (often confusingly) go to zero.

The best way to avoid problems is to avoid use of Java synchronization. One of the most common uses of synchronization is to implement a **pool** of serially reusable objects. Often, you can simply add a serially reusable object to an existing pooled object. For example, you can add a JDBC connection and statement object to the instance variables of a single thread model servlet, or you can use the Oracle JDBC connection pool, rather than implement your own synchronized pool of connections and statements.

If you must use synchronization, then you should either avoid deadlock or detect it and break it. Both strategies require code changes. So neither can be completely effective, because some system code uses synchronization and cannot be changed by the application.

To prevent deadlock, simply number the objects that you must lock and ensure that clients lock objects in the same order.

Proprietary JVM extensions may be available to help spot deadlocks without having to instrument code, but there are no standard JVM facilities for detecting deadlock.

> **See Also:** Chapter 2, "Performance and Scalability" for more information on synchronization

HA-SW-2: Resource Use

Monitor resource use and fix resource leaks.

The periodic restart strategy provides protection against slow resource leaks. But it is also important to spot applications that are draining resources too quickly, so that the buggy software can be fixed rather than restarted more frequently. Leaks that prevent continuous server operation for at least 24 hours must be fixed by programmers, not by application restart.

Common programming mistakes are:

- Not returning a resource to a pool (or not removing it from a pool) after handling an error
- Relying on the garbage collector to invoke `finalize()` and free resources

  Never rely on the garbage collector to manage any resource other than memory.
- Not discarding old object references (which prevent recycling an object's memory)

Monitoring resource usage should be a combination of code instrumentation and external monitoring utilities. With code instrumentation, calls to an application-provided interface, or calls to a system-provided interface like Oracle Dynamic Monitoring Service (DMS), are inserted at key points in the application's resource usage lifecycle. Done correctly, this can give the most accurate picture of resource use. Unfortunately, the same programming errors that cause resource leaks are also likely to cause monitoring errors. That is, you could forget to release the resource, or forget to monitor the release of the resource.

Operating system commands like `vmstat` or `ps` provide process-level information such as the amount of memory allocated, number/state of threads, or number of network connections. They can be used to detect a resource leak. Development tools like Purify or JavaProbe can be used to pinpoint the leak.

Note: In addition to compromising availability, resource leaks and overuse decrease performance. See Chapter 2, "Performance and Scalability".

HA-SW-3: Session State

Store session state in a database, indexed by session identifier (ID) stored in a cookie.

Because Jserv's HttpSession is not statesafe, it cannot be the only store for session data if you want to prevent multiple session failures in the event of the failure of a single JVM process.

The biggest problem is that an application might have session state that cannot be stored as-is in a database. For example:

- JDBC connections or statements
- Connections to other services, such as credit card verification or email
- Unencrypted sensitive data, such as passwords

To address the first two, examine the non-serializable (in the Java sense) session state and separate what is really specific to a particular client from what could be cached or parameterized, pooled, and serially reused across many clients. For example, an application might open a JDBC connection and prepare several SQL statements for each client session. When Smith logs on, the application prepares a number of SQL statements like:

```
Select x,y from T where param1 = ? and client_id = 'Smith'
```

As Smith enters different values (via his browser) for `param1`, different (personalized) values of x and y are displayed. Session state can be reduced in this case to the single (serializable) value `'Smith'` by parameterizing `client_id` in the SQL:

```
Select x,y from T where param1 = ? and client_id = ?
```

and then serially reusing the connection and its associated statements across many clients. The **serial reuse** can be accomplished with:

- An advanced JDBC connection cache

  It is necessary to cache a connection and the statements created relative to that connection together.

- A pool of application objects that encapsulates a connection and dependent statements (for example, a model object in the model, view, controller design pattern)

- A pool of single thread model servlets, each having a JDBC connection and dependent statements as instance variables

  This model gives the programmer the most control, and does not require a bug-free connection cache or use of Java synchronization to implement your own pool. Unfortunately, it precludes use of JSPs, because none of the JSP scopes map to instance variables of single thread model servlets.

When implementing statesafety, tradeoffs affect performance, scalability, and availability. If you do not implement statesafe applications, then:

- A single JVM process failure will result in many user session failures.

  Work done shopping, filling in a multiple page form, or editing a shared document will be lost, and the user will have to start over.

- Not having to load and store session data from a database will reduce CPU overhead, thus increasing performance.

- Having session data clogging the JVM heap when the user is inactive reduces the number of concurrent sessions a JVM can support, and thus decreases scalability.

  In contrast, a statesafe application can be written so that session state exists in the JVM heap for active requests, which is typically 100 times fewer than the number of active sessions.

For those concerned with the performance of statesafe applications, here are some performance optimizations:

- Minimize session state.

  For example, a security role might map to detailed permissions on thousands of objects. Rather than store all security permissions as session state, just store the role ID. Maintain a cache, shared across many sessions, mapping role ID to individual permissions.

- If you can identify a few session variables that change much more frequently than the rest, then store these attributes in a cookie to avoid database updates on most requests.
- If you can identify a set of session variables that are read on many requests, then consider using HttpSession as a cache for that session data in order to avoid having to read it from the database on every request.

  You must manually synchronize the cache, which requires care to handle planned and unplanned transaction rollback. You should also configure the load balancer to be sticky based on the HttpSession cookie value.

Note that statesafe middle tiers have been implemented by a number of high-end commercial Web sites, and they scale to extremely high workloads. A well-implemented database backing store can service thousands of state save and restore operations for each CPU.

See Also: Chapter 2, "Performance and Scalability"

HA-SW-4: Catch-All Exceptions

Discard objects that throw catch-all exceptions like SQLException, RemoteException, Error, or RuntimeException. Avoid poisoned pools.

In many cases, these exceptions indicate that the internal state of the invoked object is corrupt and that further invocations will also fail. Keep the object reference only if careful scrutiny of the exception shows it is benign, and further invocations on this object are likely to succeed.

Adopt a *guilty unless proven innocent* attitude. For example, a SQLException thrown from an Oracle JDBC invocation could represent one of thousands of error conditions in the JDBC driver, the network, or the database server. Some of these errors (for example, subclass SQLWarning) are benign. Some SQLExceptions (for example, ORA-3113: end of file on communication channel) definitely leave the JDBC object useless. Most SQLExceptions do not clearly specify what state the JDBC object is left in. The best approach is to enumerate the benign error codes that could occur frequently in your application and can definitely be retried, such as a unique key violation for user-supplied input data. If any other error code is found, then discard the potentially corrupt object that threw the exception.

Discard all object references to the (potentially) corrupt object. Be sure to remove the corrupt object from all pools, in order to prevent pools from being poisoned by corrupt objects. Do not invoke the corrupt object again. Instantiate a brand new object instead.

When you are sure that the corrupt objects have been discarded and that the catching object is not corrupt, throw a non-catch-all exception so that the caller does not discard this object, or retry the failed invocation as per "HA-SW-6: Retry Transactions Once".

HA-SW-5: Finally Clause

Always use a finally clause in each method to clean up (restore invariants to) instance variables and static variables.

Remember that in Java, it is impossible to leave the try or catch blocks (even with a throw or return statement) without executing the finally block. If for any reason the instance variables cannot be cleaned, then throw a catch-all exception that should cause the caller to discard its object reference to this now corrupt object.

If for any reason the static variables cannot be cleaned, then throw an InternalError or equivalent that will ultimately result in restarting the now corrupt JVM.

HA-SW-6: Retry Transactions Once

Retry failed transactions and idempotent HttpServlet.doGet() operations once. Otherwise, do not retry using the same method with the same parameters on the same object instance. Propagate an exception to the caller instead.

Retries are discouraged in general, because if every catch block of an N frame try..catch stack performs M retries, then the innermost method gets retried (MN)/2 times. This is likely to be perceived by the end user as a hang, and hangs are worse than receiving an error message.

If you could pick just one try..catch block to retry, then it would be best to pick the outermost block. It covers the most code and therefore also covers the most exceptions. Of course, only idempotent operations should be retried. Transactions guarantee that database operations can be retried as long as the failed try results in a rollback, and as long as all finally blocks restore variables to a state consistent with the rolled back database state. It will often be the case that a servlet's doGet method will perform retry, and a servlet's doPost method will rollback any existing transaction and retry with a new transaction.

Other cases where a retry is warranted are:

- The semantics of a checked exception suggest a retry using a different method or different parameters.

  For example, `ShoppingCart.insert()` might throw an `ItemExists` exception, and this should be caught and `ShoppingCart.incrementQuantity()` should be tried.

- Several object replicas exist (usually in different processes).

  A failure of one (for example a remote exception) could be caught and another replica could be tried. Retry only if the operation is idempotent. Most catch-all exceptions do not guarantee that all effects from the failed invocation are undone.

For example, if the database tier uses Oracle Real Application Clusters, then a new connection may be to any available database server machine that mounts the desired database. For JDBC, the `DataSource.getConnection()` method is usually configured to pick among ORAC machines.

HA-SW-7: Database Update

HttpServlet.doGet() should not update the database. HttpServlet.doPost() should use one transaction to update one database. The user must be able to query whether the update occurred or not.

The HTTP specification states that the `GET` method should be idempotent and free of side-effects. Proxies or caches along the route from client to mid-tier, as well as the client pressing the reload button, could cause the `GET` method at the mid-tier to be called 0, 1, or more times.

HTTP Post is not assumed to be idempotent. Browsers typically require client confirmation before re-POSTing, and intermediate proxies and caches do not retry or cache the result of a POST. However, a failure may require the client to manually retry (press RELOAD or press BACK and then re-submit), which is not safe unless the update is idempotent.

One way applications can warn users about potential duplicate requests is to encode a unique request-ID in a hidden form field and write the request-ID of each update request to the database. An update request first compares its request-ID with the database of already-processed requests and then warns the user about a potential duplicate. Because the user may have intended to submit two updates, the system cannot make duplicate suppression transparent. Another good practice is to label non-idempotent submit buttons with:

- Advice against reloading or resubmitting the current page
- Instructions on what application level logs to consult should a failure occur

Transactions should not span client requests, because this can tie up shared resources indefinitely.

Requests generally should not span more than one transaction, because a failure in mid-request could leave some transactions committed and others rolled back. If this requires application-level compensation to recover, then availability or data integrity may be compromised.

Transactions generally should not span more than one database, because distributed transactions lock shared resources longer, and failure recovery may require simultaneous availability and coordination of multiple databases.

Applications that require a single client request (for example, confirm checkout) to ultimately affect several databases (for example credit card, fulfillment, shopping cart, and customer history databases) should perform the first step at one database and in the same transaction enqueue a message at the first database addressed to the second database. The second database will perform the second step and enqueue the third step, and so on. This queued transaction chain will eventually complete automatically, or an administrator will see an undeliverable message and will have to manually compensate.

Periodic Operations Procedures

The preceding sections covered provisioning Web-site hardware and software for availability. Operations staff also play an important role. First, they must carry out certain preventative procedures, primarily backups and application server process restarts, on a regular basis. Second, they must be prepared to react to failures in order to make the situation better, not worse. Finally, they must be able to react to a constantly changing environment, with new hardware for increased capacity and with frequent software and Web content updates to support the fast-paced online world.

HA-OP-1: Backup Schedule

Implement a database and operating system file backup schedule that supports restoring the entire Web site to brand-new computers, storage, and network equipment with less than 24 hours combined downtime and data loss.

This provides a last line of defense against the worst (but not all that uncommon) failures, such as careless employees, botched upgrades, security break-ins, and software bugs that corrupt stored data.

It is important to back up everything, including router, firewall, and load balancer configuration, operating systems and their configuration, Oracle9*i*AS software and its configuration. But the backend Oracle database should be where most of your data is stored; it should receive the most attention.

To ensure that backed up files are consistent with database data, and to further leverage the database backup and restore features, consider storing master copies of flat files in Oracle Internet File System. Because Oracle Internet File System is a database application, your master files are backed up the same way as your database. You can use FTP to copy files from Oracle Internet File System to the mid-tier file systems. On NT, you can simply create a drive letter for Oracle Internet File System. For better performance and availability, it is better to copy the mid-tier files to the local operating system file system (or cache them in Web cache) than to access Oracle Internet File System each time a file is requested.

If a failure requiring restoration of the database from a backup happens once a year and takes 24 hours to fix, then availability is 99.7%. Using a physical standby database can reduce the database restore time to a few minutes. The standby database is always in recovery mode, and its state lags the primary database by some fixed period (for example, 15 minutes). When the primary fails, applications switch to the standby. The hard part here is to detect the failure, which could be a corrupt infrequently accessed disk block, accidentally run batch job, or other non-obvious failure. It must be detected within the fixed period lag, before it possibly gets propagated to the standby.

HA-OP-2: Server Restarts

Restart Java servers daily or according to measured frequency of failure.

Many hard-to-debug software failures are caused by resource leaks in serially reusable servers that gradually degrade performance. Degradation can become so severe for some Java server processes that the JVM process stops responding, even to kill commands. Resource leaks may also occur in the Apache Web server processes, especially if a lot of application code runs there, using mod_perl or mod_php, for example. Usually, the Apache server will not fail often enough to impact overall availability, but restarting it affords an opportunity to rotate the access and error log files.

Resource leaks can cause failure of other software servers, such as:

- Directory, naming, and security servers
- Databases
- Firewalls and load balancers
- Operating systems, especially less mature or non-Unix variants

Availability of these servers, however, should be high enough not to warrant periodic restart.

Determining a good restart frequency for Apache and Jserv processes may take some trial and error. Begin with a daily restart and record all Java process crashes and hangs. If many failures are observed, then increase the frequency of restart slightly. But also attempt to localize the faulty software module (using dumps, traces, or error logging) and insist that the bugs be fixed. If very few failures are observed, then decrease the frequency of restart.

To periodically restart all Web servers and their associated Jserv servers in a cluster without causing any client requests to fail, cycle through all Web servers and for each Web server W:

1. Issue the "out-of-service W" command to the load balancer.
2. Perform a graceful restart on W.
3. Restart local Jservs on W.
4. Issue the "in-service W" command to the load balancer.
5. Repeat with next W.

This approach should scale to as many Web servers as most load balancers can handle. For example, if you need to restart Jservs every twelve hours or things start failing, and if a restart and warm-up of Apache and Jserv processes on a single machine takes ten minutes, then you can scale to 72 mid-tier machines.

These regular maintenance periods can also be used to refresh the configuration, code, and static content files from a master copy, archive and compress mid-tier logs, and so on.

HA-OP-3: Mid-Tier File Synchronization

Synchronize mid-tier files by taking one computer out of the cluster at a time, if possible. Otherwise stage file updates so the entire cluster goes offline for just seconds to synchronize.

File system content should be synchronized across mid-tier computers. Depending on configuration settings, some files can be changed without restarting Apache or Jserv processes, while other changes require a restart. Some files can be NFS mounted, while other files cannot. Some applications can behave erratically if new class files are dynamically loaded and mixed with old class files. In short, making a shared or networked file system highly available while changing files on a running mid-tier computer is full of unknowns and worry.

A good practice instead is to exploit the incremental offline capability afforded by the cluster load balancer to synchronize a mid-tier computer's local files with a master copy while the mid-tier computer is offline. The challenge of such an incremental offline synchronization is to move all mid-tier computers from version N of synchronized file system content to version N+1, while making sure that all requests for a given session are directed to mid-tier computers with the same version of file system content.

You can configure the load balancer to use sticky routing, so that all requests within a session are routed to the same mid-tier computer (provided it is still online). Thus, during the period of rolling synchronization, session failover should be defeated or extended. For example, you could store the synchronization version number as part of the session ID and check it on each request. This will prevent errors reading or writing session data, in case its datatype changes from one version to the next.

As version N+1 computers come online, be sure to monitor them. If something goes wrong, then you can stop the synchronization, leave unmodified version N computers online, and take the troublesome version N+1 computers offline and fix the problem.

Finally, note that some configuration, code, or content changes cannot be performed one mid-tier server at a time. For example, version N and N+1 of an application may not both be able to run against the same database schema. It is always good practice to have a method to update all servers in a cluster, so that they are all consistent and with minimal downtime. One strategy is, first stage the updates to a separate directory on each mid-tier computer. Second, go offline just long enough to swap in the updates (for example, with a symbolic link). Third, restart the servers. To minimize downtime, the swap and restart should happen in parallel for each server in the cluster.

HA-OP-4: Document Procedures

Document all recovery and repair procedures, and practice them regularly.

During a failure, operations staff will be under pressure. It may be difficult to think clearly. It is not a good time to try a new, risky, or unfamiliar repair procedure. Document all of the following:

- Backup files and archived log locations
- Diagnostic tool syntax
- Location of spare parts (disks, network cards)
- How to replace failed parts (what should be powered off/on, what racks, chassis, or slots hold what components)
- What needs to be restarted
- Contacts (supervisors, support personnel, other experts)

Conduct periodic fire drills. When a real failure occurs, it will not be the first time the staff has had to react under pressure. Try to simplify complex repair procedures to minimize errors.

2

Performance and Scalability

This chapter provides a checklist of what we consider performance and scalability best practices in designing, implementing, tuning, and testing your Web application that is to be deployed on Oracle9*i*AS Release 1 (v1.0.2.2). It is based on the collective experience and work of the PDC performance group. Some of the checklist items are based on results of our performance tests on Oracle9*i*AS. Others are based on common performance issues we encountered while working with our customers who are building and deploying their Web application on Oracle9*i*AS.

This chapter contains these topics:

- Performance and Scalability Overview
- Servlets and JavaServer Pages: A Case Study
- Servlets and JavaServer Pages: Design and Implementation
- Servlets and JavaServer Pages: Deployment
- Java Performance
- Oracle HTTP Server Tuning
- Performance Testing
- Dynamic Monitoring Service (DMS)
- Code Example
- References and Resources

Performance and Scalability Overview

Some developers suggest you should make it work, then make it fast. This is common advice by those who think one should avoid any premature optimization that would yield little benefit. It really should not be taken literally that performance and scalability should be deferred until the application is in production.

We have had cases of customers who were experiencing performance problems days before their system was ready to go live. Some common complaints were that their system was unable to support even a small number of concurrent users, their response time was unacceptable, or there was a memory leak in the system which degraded performance. On some occasions, we were able to help them by making some configuration suggestions. Other times, we found the problems were so ingrained in the application's architecture that there were no simple solutions we could give them other than suggesting that they redesign the system.

This chapter focuses on applications written in Java using servlets and JavaServer Pages (JSPs). It includes sections on performance and tuning tips for servlets and JSPs deployed on JServ, Java performance tips, configuration tips for the Oracle HTTP Server, and some suggestions on performance testing.

Some of the performance and tuning tips for servlets, JSPs, and the Oracle HTTP Server are from the *Oracle HTTP Server Performance Guide*, available in the Oracle9*i* Application Server Platform-specific Documentation. We include examples as well as results of our performance tests on these practices.

This chapter is intended for application designers and system administrators who are already familiar with Java, servlets, JSPs, and Apache.

Performance Tradeoffs

It is critical to consider performance when designing your Web application, but it is also important to remember that there are tradeoffs between performance, availability, security, and flexibility. For more information on availability and security, you can refer to Chapter 1, "Availability of Java Applications" and Chapter 4, "HTTP Security". Here, the focus is the tradeoff between performance and flexibility.

For example, in Java you can define a class as final to improve performance. However, a final class cannot be subclassed; so you will lose some flexibility. Another example is the use of an object factory to centralize object creation instead of creating the object directly. The object factory will undoubtedly add performance

overhead to the application, because of the extra method calls, but it allows you more flexibility.

Balancing these tradeoffs is not always easy; often it is the subject of debate. Also, a performance gain in one area can be a performance loss in another. Our suggestions are:

- Understand why and when you should use each performance technique.
- Evaluate the performance gains from each technique against your performance goals and design criteria.
- If there are any conflicts or tradeoffs, then ask if the performance gains justify the cost in flexibility or maintainability.

Servlets and JavaServer Pages: A Case Study

Servlets and JavaServer Pages (JSPs) are used on the server side to access data and to generate dynamic content in a Web application. Their framework makes it possible to build Web applications quickly and easily. The disadvantage is that their flexibility can make debugging very difficult. JSPs can be especially difficult, because you can easily mix HTML with Java code in the same page.

The architecture does matter when you design your Web application. If it has a fundamental problem, then there is usually very little tuning you can do to improve performance significantly without redesigning the system.

We use the following customer application as a case study. The application used JSPs. The user logged in through a home page and was authenticated. If the authentication succeeded, then the user's profile was retrieved from the database, and the home page forwarded the user to the appropriate menu page. This is a common flow of operation in many Web applications.

The following is a high level description of the original architecture:

1. Each JSP accessible by the user (such as the home page) calls a `beginRequest()` method to check if the user has been authenticated based on the cookie information from the user.
2. If the user is valid, then an entry will be added to a Hashtable with the user's profile object, using the thread identifier (ID) as the key. It will also obtain other resources that the user may need and store them in this profile object. One of these resources can be a JDBC connection.
3. The JSP will do whatever data query or access it needs by calling the appropriate method to generate the content. It may include many nested JSPs.

4. Any method that needs access to the user profile information will need to go to the Hashtable to get the user profile and any resource objects using the thread ID.
5. Before returning the content to the user, the JSP will call an `endRequest()` method to remove the entry from the Hashtable and release the profile object and resources. The user's profile or any state information is encoded in a cookie and returned to the user, so subsequent requests from the same user can parse the cookie for the user information. No state information is kept in the server using HttpSession or session scope beans.

This architecture has several design and Java performance problems:

- User Information
- Use of Hashtables
- endRequest() Called Manually

User Information

User information in the cookie has to be parsed, and the user has to be reauthenticated in each request.

The application used this architecture to appear **stateless**. User information is encoded in a cookie and sent back to the user at the end of each request. Each JSP calls the `beginRequest()` method to parse the cookie and reconstruct the user profile object from the cookie before processing each request.

Problems:

- Parsing the cookie again for each request generates tremendous overhead, because it involves mostly String operations.
- Reauthenticating the user again for every request is expensive, because it requires repeated access to the database.
- Many new objects have to be created and destroyed in each request.

Use of Hashtables

High-cost Hashtables are used to store objects that will never be accessed concurrently.

The implementation in the `beginRequest()` stores the user and request-related information in static Hashtables using the Java thread ID as the key. The objects are stored in the Hashtables at the beginning of the request and removed at the end of the request. Any operations that need to use the resources or access the user data use the Java thread ID to retrieve these objects while processing the request.

Problems:

- Each access to these objects has to pay the synchronization overhead, even though objects stored in these Hashtables can never be accessed concurrently by another thread.
- The method calls `put()`, `get()`, and `remove()` are constantly active in these Hashtables, because each element will remain in them for only a short time.

endRequest() Called Manually

The `endRequest()` must be called manually to release the resources before exiting the JSPs. Problems:

- Failure to call `endRequest()` to remove the objects and free the resources from the Hashtables can result in a major memory leak.

  The JSP developer must understand the flow of operations between the JSPs and decide which JSP should make the `endRequest()` call.

- We found many JSPs in this application were in fact not releasing valuable resources (such as JDBC connections) because the `endRequest()` method was not called accordingly.

  Therefore, many of the connections could not be reused, and objects were not released properly, causing gradual memory leaks.

We are not suggesting that making an application stateless is a bad practice. Stateless applications are more highly available and secure, but you pay a price in performance.

We rewrote part of the application by using many of the best practices suggestions in this chapter, and we were able to improve performance without giving up any functionality. But the changes we made were not trivial. In order to get a significant performance gain, we needed to redesign part of the architecture and rewrite much of the Java and JSP code.

We use this example to stress the importance of selecting the right architecture before implementing your application. If you wait until the system is in production before you uncover these types of performance problems, then it may be too late to fix them. It will certainly cost more than getting it right the first time.

Servlets and JavaServer Pages: Design and Implementation

This section contains our design and implementation recommendations for using servlets and JSPs. Because the JSP is an extension of the servlet, some of these best practices are applicable to both.

Oracle9*i*AS supports servlets and JSPs through Oracle JavaServer Pages (OJSP) and JServ. The performance tips in this section are specific to using JServ and OJSP through the Oracle HTTP Server.

Note: Because JServ is based on Servlet 2.0, we will not include any best practices in this section that require any Servlet 2.2 support.

PERF-1: Storing State Information

Use session to store state information that need not be persistent but can be recovered easily.

A real-world Web application can rarely be totally stateless. Most Web applications have to service a sequence of requests from the same user with certain state information passed between requests. When a Web application is called stateless, it usually means the application takes the extra effort to ensure the state information is either persistent or recoverable in the event of a process failure on the server side. Though it would be nice to make a Web application truly stateless to improve availability, it is generally costly to do so.

The servlet and the JSP runtimes support session tracking. State information can be saved in the session objects associated with a session ID. As long as the session has not expired, a client who comes back to the same JServ process with the session ID can retrieve any data that is stored in the HttpSession object. The data stored in the session will be released either when the application explicitly calls `invalidateSession()` or when the session times out based on the session timeout parameter.

The most commonly cited drawbacks with using sessions are:

- Session data is not persistent when the session times out or the process fails.
- Sessions increase memory usage.
- The client's browser must accept cookies.

Our performance tests have shown, however, that using sessions for certain types of data can minimize these problems and provide significant performance gains.

In our case study, we found the application's user profile information can take advantage of using sessions because:

- It was relatively small, so the memory cost was low.
- It was costly to re-establish in every request.
- It was needed only for the duration of the session.
- It did not need to be persistent.
- It could be recovered easily in case of process failure.

We modified the application in the case study by using a session scope bean to store the user profile information when the user logged in from the home page. This eliminated some of its performance problems:

- We did not parse the cookie or reauthenticate at each request.

  If the session was still valid when the user returned with the next request, then we skipped the step of parsing the cookie or reauthenticating, unless the request required update or access to restricted data. Only in the case of session failure did we have to parse the cookie to recreate the user profile object.

- By using a session scope Java bean, which can pass the user information from page to page while processing the request, we eliminated the use of all Hashtables.

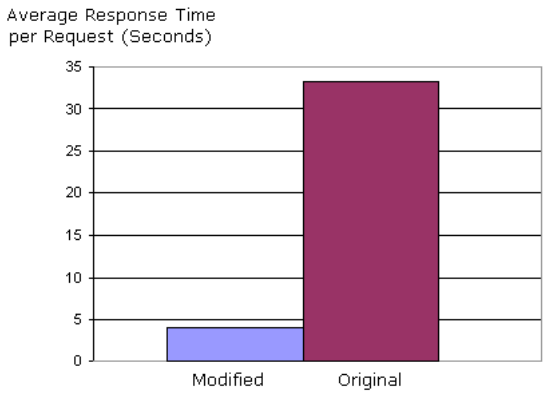
- We did not have to make any explicit `endRequest()` method call in the JSP.

  Because we used a session scope bean, we called `invalidateSession()` explicitly to end the session if the user logged out. Otherwise, we simply let the session time out.

After implementing these changes and others we suggest in this chapter, we ran a performance test comparing the original and modified versions, using exactly the same configuration. The test simulated a set of seven requests per user. Each user logged in, selected some functions from the menu, and logged out. The test was run with 100 concurrent users who repeatedly executed these seven requests for a

duration of one hour. Our modified version ran about *eight times faster* than the original application , as shown in Figure 2–1.

Figure 2–1 Case Study Performance Results



The memory overhead for keeping the user profile in session in our modified version was only about 1MB for 12,000 sessions. This is a relatively small overhead compared to the significant performance gain.

We suggest the following guidelines when using sessions:

- Store certain state information as HttpSession objects (in servlets) or using session scope Java beans (in JSPs), if it does not have to be persistent but is costly to re-establish for every request during the session.
- Do not store any shared resource objects in the session objects or session beans.

  Objects that are stored in the session objects will not be released until the session times out. If you hold any shared resources that have to be explicitly released to the **pool** before they can be reused (such as a JDBC connection), then these resources may never be returned to the pool properly and can never be reused.
- Set the session timeout parameter appropriately to support the expected usage of the application.
- Monitor the memory usage for the data you want to store in session objects.

Make sure there is sufficient memory for the number of sessions created before the sessions time out.

You can use any platform-specific utilities (such as ps top on Solaris) to monitor the memory usage of a JServ process. Dynamic Monitoring Service (DMS) also provides additional built-in metrics that you can use to display the memory usage of the JVM in each JServ process.

PERF-2: Connection Pooling and Caching

Use JDBC connection pooling and connection caching.

Constant creation and destruction of resource objects can be very expensive in Java. In "Java Performance" on page 2-21, we suggest using a resources pool to share resources that are expensive to create. The JDBC connections are one of the most common resources used in any Web application that requires database access. They are also very expensive to create. We have seen overhead from hundreds of milliseconds to seconds (depending on the load) in establishing a JDBC connection on a mid-size system with 4 CPUs and 2 GB memory.

In JDBC 2.0, a connection-pooling API allows physical connections to be reused. A pooled connection represents a physical connection which can be reused by multiple logical connections. When a JDBC client obtains a connection through a pooled connection, it receives a logical connection. When the client closes the logical connection, the pooled connection does not close the physical connection. It simply frees up resources, clears the state, and closes any statement objects associated with the instance before the instance is given to the next client. The physical connection is released only when the pooled connection object is closed directly.

The term pooling is extremely confusing and misleading in this context. It does not mean there is a pool of connections. There is just one physical connection, which can be serially reused. It is still up to the application designer to manage this pooled connection to make sure it is used by one client at a time.

To address this management challenge, Oracle's extension to JDBC 2.0 also includes connection caching, which helps manage a set of pooled connections. It allows each connection cache instance to be associated with a number of pooled connections, all of which represent physical connection to the same database and schema. You can use one of Oracle's JDBC connection caching schemes (dynamic, fixed with no wait, or fixed wait) to determine how you want to manage the pooled connections, or you can use the connection caching APIs to implement your own caching mechanisms.

See Also:

- "Code Example" on page 2-37 for an example of a servlet using connection pooling and caching
- For a detailed description and additional examples of connection pooling and connection caching, see the *Oracle8i JDBC Developer's Guide and Reference* in the Oracle Application Server Documentation Library

PERF-3: Statement Caching

Use JDBC statement caching.

Use JDBC statement caching to cache a JDBC `PreparedStatement` or `OracleCallableStatement` that is used repeatedly in the application to:

- Prevent repeated statement parsing and recreating
- Reduce the overhead of repeated cursor creation

The performance gain will depend on the complexity of the statement and how often the statement has to be executed. Since each physical connection has its own statement cache, the advantage of using statement caching with a pool of physical connections may vary. That is, if you execute a statement in a first connection from a pool of physical connections, then it will be cached with that connection. If you later get a different physical connection and want to execute the same statement, then the cache does you no good.

See Also:

- "Code Example" on page 2-37 for an example of a servlet using connection pooling and caching
- For a detailed description and additional examples of connection pooling and connection caching, see the *Oracle8i JDBC Developer's Guide and Reference* in the Oracle Application Server Documentation Library

PERF-4: Mid-Tier Caching

Use mid-tier caching mechanisms Oracle9iAS Web Cache and Oracle9iAS Database Cache to cache commonly shared or reused data.

Caching commonly shared data in the mid-tier can provide significant performance improvement. Oracle9*i*AS Web Cache and Oracle9*i*AS Database Cache can be used in the mid-tier to cache HTML pages, large shared objects, or data.

Oracle9*i*AS Web Cache can store both static and dynamically generated Web content. Similarly, Oracle9*i*AS Database Cache can store frequently used data and stored procedures. By storing frequently accessed pages, data, or procedures in memory, you no longer need to go through the Oracle HTTP Server, and you get significant performance improvements.

Oracle9*i*AS Web Cache can be added as a front end to the Oracle HTTP Server during deployment and does not require any application code changes. However, you may be able to get additional benefit if you design your application to take advantage of its particular features and functions.

These components have very different performance implications depending on how they are deployed and configured.

See Also:

- *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library
- *Oracle9iAS Database Cache Concepts and Administration Guide* in the Oracle9*i* Application Server Documentation Library

PERF-5: Database Connections

Avoid using more than one database connection simultaneously in the same request.

Using more than one database connection simultaneously in a request can cause a deadlock in the database. This is most common in JSPs. First, a JSP will get a database connection to do some data accessing. But then, before the JSP commits the transaction and releases the connection, it invokes a bean which gets its own connection for its database operations. If these operations are in conflict, then they can result in a deadlock.

Furthermore, you cannot easily roll back any related operations in case of failure, if they are done by two separate database connections.

Unless your transaction spans multiple requests or requires some complex distributed transaction support, you should try to use just one connection at a time to process the request.

PERF-6: Database and SQL Tuning

Tune the database and SQL statements.

Current Web applications are still very database-centric. From 60% to 90% of the execution time on a Web application can be spent in accessing the database. No amount of tuning on the mid-tier can give significant performance improvement if the database machine is saturated or the SQL statements are inefficient.

Monitor frequently executed SQL statements. Consider alternative SQL syntax, use PL/SQL or bind variables, pre-fetch rows, and cache rowsets from the database to improve your SQL statements and database operations.

> **See Also:** *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation

PERF-7: Memory Leaks

Avoid common errors that can result in memory leaks.

In Java, memory bugs often appear as performance problems, because memory leaks usually cause performance degradation. Because Java manages the memory automatically, developers do not control when and how garbage is collected. To avoid memory leaks, your applications must:

- Release JDBC ResultSet, Statement, or connection.

  Release failures here are usually in error conditions. Use a finally block to make sure these objects are released appropriately.

- Release instance or resource objects that are stored in static tables.

  The original application in the case study was typical of this type of memory leak. The profile object and other resource objects were stored in static Hashtables, to be released at the end of a JSP by calling the `endRequest()` method. But error conditions in any of the nested JSPs or forwarded JSPs would leave these objects in the Hashtables, causing memory leaks. This design is too error prone, because each JSP designer must understand the flow of operation between JSPs and know whether the `endRequest()` has to be called in the page.

- Perform clean up on serially reusable objects.

  One application we helped review was appending error messages to a Vector defined in a serially reusable object. The application never cleaned the Vector before it was given to the next user. As the object was reused over and over again, error messages accumulated, causing a memory leak that was difficult to track down.

PERF-8: Spawning Threads

Avoid spawning threads in the application.

With the infrastructure provided by the servlet engine, a Web application rarely needs to spawn its own threads. Improper management of user-spawned threads can create many runaway threads, causing system deadlocks as well as memory leaks.

We saw one case in which the application was spawning a thread each time it wrote to a user-defined log file, so the client did not have to wait for the expensive log operation to complete before receiving the response. Unfortunately, the implementation failed to handle some exception conditions. When these exception conditions were encountered, it created thousands of runaway threads to retry the log operation, causing huge memory leaks and performance degradation.

PERF-9: SingleThreadModel in Servlets

Consider using SingleThreadModel in servlets as a resource pool and to minimize synchronization if appropriate.

The SingleThreadModel implementation in the servlet engine is similar to maintaining a pool of serially reusable servlet instances, where each servlet instance can be used by only one thread at a time. Therefore, any servlet which implements the SingleThreadModel is considered to be thread-safe, and no synchronization is needed when accessing any servlet instance variables or objects.

This is effectively getting the support of a managed pool without the overhead of implementing it yourself. In one of our performance tests, we have seen performance improvement of up to 25% under load by using SingleThreadModel to reduce synchronization and manage reusable objects.

If your application uses a resource that is expensive to create, and you would like to be able to reuse it serially, then you can implement your application by using a SingleThreadModel in a servlet. You can create the resource in the servlet's `init()` method and save it as an instance object. Since the SingleThreadModel is

thread-safe, the resource object can be used by only one thread at a time. You can then release the object in the servlet's `destroy()` method when the instance is to be shut down.

The main difference between using SingleThreadModel to pool resources and implementing your own pool is flexibility. With SingleThreadModel, the number of objects you can create is controlled by the maximum number of instances per process multiplied by the maximum number of JServ processes.

For example, if you use SingleThreadModel with 10 instances per JServ process and there are 10 JServ processes, you can end up creating 100 instances of a resource. These resources are usually not released until the instance's `destroy()` method is called. So SingleThreadModel is not as flexible as implementing your own pool, but it is simple to use and can provide good performance gains for the appropriate applications.

Examples where SingleThreadModel is *not* appropriate:

- You will be using a JSP as a front end to the servlets (see "PERF-13: Thread-Safe JSPs" on page 2-16).
- Your application has high variation between busy and idle times.

  Resources would be wasted when the system is idle.

Consider using SingleThreadModel if:

- You want your application to create expensive resources that can be serially reused, but you do not want to create your own pool to manage them.
- Your application executes frequently.
- You want your application to be thread-safe, and you want to limit the overhead of synchronization.

If you are using SingleThreadModel:

1. Go to the `jserv.properties` file
2. Determine the value of `security.maxConnections`
3. Go to the `zone.properties` file
4. Set `singleThreadModelServlet.maximumCapacity` to be greater than or equal to the value determined in step 2

PERF-10: Servlets Startup Operations

Perform any costly one-time startup operations in the servlet init() method.

Use the servlet's `init()` method to perform any costly one-time initialization. Examples include:

- Setting up resource pools
- Retrieving common data from the database that can be cached in the mid-tier to reduce warm-up time

See Also: "Code Example" on page 2-37

PERF-11: Separate JSPs

Separate presentation logic from business and data access logic in JSPs.

Because it is possible to include any Java code in the JSPs, it is easy to mix business and data access Java code with the presentation logic in the JSPs. Including business and data access logic in the JSPs makes the application much more difficult to maintain and debug. It increases the footprint of the JSPs and limits the efficiency of resource use and management.

For example, if each JSP has its own Java code to handle database connections and data access, it will be much more difficult to allow the database connections to be reused or shared when processing a request and between requests. Because a database connection is an expensive resource, sharing failures can have significant performance impacts.

PERF-12: Static or Dynamic Include in JSPs

Choose static or dynamic include appropriately.

The JSP has two different include mechanisms:

- Static include uses page directive:

  ```
  <%@ include file="filename.jsp" %>
  ```

- Dynamic include uses page directive:

  ```
  <jsp:include page="filename.jsp" flush="true" />
  ```

Static include creates a copy of the file in the JSP. Therefore, it increases the page size of the JSP, but it avoids additional trips to the request dispatcher.

If you use static includes, keep files below the 64K limit of the service method of the generated page implementation class. If you need to work around this limitation, see the section "Workarounds for Large Static Content in JSP Pages" in *Oracle JavaServer Pages Developer's Guide and Reference* at `http://otn.oracle.com/docs/index.htm`.

Also, even if you set `developer_mode=true` in the `zone.properties` file, changes made to a static include file in a running system may not be reflected until you update the main JSP or delete the class file generated for the main page.

Dynamic include is analogous to a function call. Therefore, it does not increase the page size of the JSP. But it does increase the processing overhead, because it must go through the request dispatcher.

Dynamic includes are useful if you cannot determine which page to include until after the main page has been requested. Note that a page that can be dynamically included must be an independent entity, which can be translated and executed on its own.

PERF-13: Thread-Safe JSPs

Do not use isThreadSafe="false" if you want multithread support when using OracleJSP. Make your JSPs thread-safe if possible.

If your JSPs are not thread-safe, then you can use the JSP directive `isThreadSafe="false"` so that they can be translated into servlets which implement SingleThreadModel. However, the current version of OracleJSP does not implement an instance pool (though this may change in future releases). At the current time, OracleJSP guarantees thread-safety by serializing requests to the same page if `isThreadSafe ="false"` is set for the page. This can have a significant impact on performance.

We ran a simple JSP test, which uses a Java bean to query the database and return a small set of data. With 100 concurrent users, we found the performance difference of using `isThreadSafe="false"` is over 50%. This number can be higher if the request has to perform more complex operations.

You should make your JSPs thread-safe and avoid using `isThreadSafe="false"` if possible. If you are using a JSP as a front end to servlets in your application, then you should also make sure your servlets are thread-safe and avoid using SingleThreadModel in your servlets.

One of the ways to make your JSPs thread-safe is proper choice of variables. You can declare variables in JSPs two ways:

- Member variables use the syntax

```
<%! long startTime=0; %>
```

- Method variables use the syntax

```
<% long startTime=0; %>
```

Member variables are declared at the class level in the page implementation class during the translation. Therefore, if you use member variables in your JSP, the JSP will *not* be thread-safe unless these member variables are themselves thread-safe.

Method variables are local to the service method of the page implementation class. Therefore, they *are* thread-safe. Try to use method variables and make your JSPs thread-safe if possible. For more information on the differences between member variables and method variables, see *Oracle JavaServer Pages Developer's Guide and Reference* at `http://otn.oracle.com/docs/index.htm`

PERF-14: Sessions in JSPs

Set session="false" if you do not use sessions in JSPs.

The default for JSP is `session="true"`. If your JSPs do not use any session, you can set `session="false"` to eliminate the overhead of creating and releasing these internal sessions created by the JSP runtime:

```
<%@page session="false" %>
```

PERF-15: Use Forward in JSPs

Use forward instead of redirect if possible.

You can pass control from one page to another by using forward or redirect, but forward is always faster. When you use forward, the forward page is invoked internally by the JSP runtime, which continues to process the request. The browser is totally unaware that such an action has taken place. When you use redirect, the browser actually has to make a new request to the redirect page. The URL shown in the browser will be changed to the URL of the redirect page in a redirect, but it will stay the same in a forward.

Redirect is always slower than forward. In addition, all request scope objects will be unavailable to the redirect page, because it constitutes a new request. Use redirect only if you want the URL to reflect the actual page that is being executed, in case the user wants to reload the page.

PERF-16: JSP Buffer

Disable JSP buffer if you are not using it.

In order to allow part of the body to be produced before the response headers are set, JSPs store the response body in a buffer. When the buffer is full or at the end of the page, the JSP runtime will send all headers that have been set, followed by any buffered body content. This buffer is also required if the page uses dynamic contentType settings, forwards, or error pages.

The default size of a JSP buffer is 8 KB. If you need to increase the buffer size, for example to 20KB, you can use the following JSP attribute and directive:

```
<%@page buffer="20kb" %>
```

If you are not using any JSP features that require buffering, you can disable it to improve performance. Memory will not be used in creating the buffer, and output can go directly to the browser. You can disable buffering by:

```
<%@page buffer="none" %>
```

Servlets and JavaServer Pages: Deployment

This section contains our deployment recommendations for using servlets and JSPs. Because the JSP is an extension of the servlet, some of these best practices are applicable to both.

Oracle9*i*AS supports servlets and JSPs through Oracle JavaServer Pages (OJSP) and JServ. The performance tips in this section are specific to using JServ and OJSP through the Oracle HTTP Server.

Note: Because JServ is based on Servlet 2.0, we will not include any best practices in this section that require Servlet 2.2 support.

PERF-17: Multiple JServs

Configure multiple JServs to improve performance and scalability.

The default configuration of Apache and JServ is to start one JServ process automatically upon start up of the Apache server, so only one JServ process is created for each host at a time. If the JServ process dies because of a JVM error, such as running out of memory, a new one will be started automatically by the monitoring mechanism in mod_jserv.

You can change the configuration to start more than one JServ process manually, so requests will be routed to the additional processes to balance the load. If the request has a session ID, it will be routed back to the same process if the session is valid. However, the drawback of manual mode is that you must restart the JServ process in case of process failure.

Our performance tests have found that in general, it is better to have more JServ processes with fewer threads than to have just one JServ with many threads. Since most Web applications have some level of synchronization in their implementation, it is better if the requests are spread out among different processes.

However, using more processes means using more memory because of the fixed memory overhead of the JVM. Also, each JServ process has to duplicate any commonly shared data in its own JVM.

Nevertheless, we found the performance gains still outweigh the memory usage in most cases. Furthermore, having more than one JServ improves availability in case of process failure.

The amount of performance gains and the number of JServ processes you should use depend on your system configuration and application. Our general recommendation is to configure two JServ processes per CPU (if you have sufficient memory) and limit the maximum number of connections per JServ to between 10 and 20.

Oracle9*i*AS provides scripts and information on starting and shutting down JServ processes in manual mode. Check the Oracle9*i*AS release notes or documentation for more information.

PERF-18: Java Heap Size

Set the Java initial and maximum heap size to meet your application needs.

Java performance degrades when it runs out of available heap. We found it is best not to let Java's free memory go below 25% at any time.

The Oracle9*i*AS default initial heap size for JServ is set to 64MB, and the maximum heap size is 128MB. This may not be sufficient for your application. You should determine your application's memory requirement for the duration of the JServ's process up time and set these values accordingly.

You can use any platform-specific utilities (such as ps, top on Solaris) to monitor the memory usage of a JServ process. Also, Dynamic Monitoring Service (DMS) provides additional built-in metrics that you can use to display the memory usage of the JVM in each JServ process. See "Dynamic Monitoring Service (DMS)" on page 2-34 for more information.

PERF-19: JServ autoreload.classes

Set autoreload.classes=false in a production system.

The default value of the autoreload.classes in the JServ's `zone.properties` file is set to `true`. This means that each time one of that zone's servlets is requested, every class that has been loaded from a repository in that zone is checked to see if it has been modified and has to be reloaded. This feature is usually more useful in a development environment than in a production system. You should set `autoreload.classes=false` if you do not need this option.

PERF-20: Preload Servlets

Preload servlets at startup time.

Use the `servlets.startup` parameter in the JServ `zone.properties` file to include any servlets that you want to be loaded upon startup of the JServ process. This will cause each servlet's `init()` method in the `servlets.startup` list to be executed even before the first request is routed to the process.

PERF-21: Pre-translate JSPs

Pre-translate JSPs before deployment.

You can use Oracle's `ojspc` tool to pre-translate JSPs and avoid the translation overhead that otherwise has to be incurred when they are first executed. You can pre-translate JSPs on the production system or before you deploy them. Also, pre-translating JSPs allows you the option to deploy only the translated and compiled class files, if you choose not to expose the JSP source files.

> **See Also:** *Oracle JavaServer Pages Developer's Guide and Reference* at `http://otn.oracle.com/docs/index.htm`

PERF-22: JSP debug flag and developer_mode

Turn off debug flag and developer_mode in the production system.

The `debug` and `developer_mode` parameters in the OracleJSP configuration can have a significant effect on performance. The `debug` flag is used to display debug information. The `developer_mode` flag is used to inform the JSP runtime whether it should automatically recompile and reload any JSPs that have changed since they were loaded. These parameters are useful mostly during development.

In a test using JDK 1.2 with 50 users, 128 MB heap, and the default TCP settings, the performance gains with `developer_mode` off were 14% in throughput and 28% in average response time.

Since the default for `developer_mode` is set to `true`, you should set it to `false` in the production system if you do not need this option. You can set `debug` and `developer_mode` to `false` in a production system in the `zone.properties` file as follows:

```
servlet.oracle.jsp.JspServlet.initArgs=debug=false,developer_mode=false
```

PERF-23: Dynamic Monitoring Service

Use DMS to monitor the performance of the JServ processes.

See "Dynamic Monitoring Service (DMS)" on page 2-34 for more information.

Java Performance

The "References and Resources" section at the end of this chapter lists many books and articles on the topic of Java performance. We do not intend to duplicate all that information in this section; instead we focus on those tips which have the greatest performance impact. The best practices in this section are based on common performance problems that we frequently see in our customers' applications.

PERF-24: Synchronization

Minimize synchronization.

Many performance studies have shown a high performance cost in using synchronization in Java. Improper synchronization can also cause a deadlock, which can result in complete loss of service because the system usually has to be shut down and restarted.

But performance overhead cost is not sufficient reason to avoid synchronization completely. Failing to make sure your application is thread-safe in a multithreaded environment can cause data corruption, which can be much worse than losing performance. These are some practices that you can consider to minimize the overhead:

- *Synchronize critical sections only.*

If only certain operations in the method must be synchronized, use a synchronized block with a mutex instead of synchronizing the entire method.

```
private Object mutex = new Object();
…
private void doSomething()
{
  // do things that do not have to be synchronized
  …
  synchronized (mutex)
  {
    …
  }
  …
}
```

- *Do not use the same lock on objects that are not to be manipulated together.*

Every Java object has a single lock associated with it. If unrelated operations within the class are forced to share the same lock, then they have to wait for the lock and must be executed one at a time. In this case, define a different mutex for each unrelated operation that requires synchronization.

```
public class myClass
{
  private static myObject1 myObj1;
  private static mutex1 = new Object();
  private static myObject2 myObj2;
  private static mutex2 = new Object();
  …
  public static void updateObject1()
  {
    synchronized(mutex1)
    {
      // update myObj1 …
    }
  }
```

```
  public static void updateObject2()
  {
    synchronized(mutex2)
    {
      // update myObj2 …
    }
  }
  …
}
```

Also, do not use the same lock to restrict access to objects that will never be shared by multiple threads. In the case study, for example, the original application used Hashtables to store objects that would never be accessed concurrently, causing unnecessary synchronization overhead.

- *Do not use multiple locks if the objects are always manipulated together.*

Similarly, if the objects are always manipulated together, make sure they share the same lock.

- *Use double-checking to reduce synchronization, such as in initialization.*

```
public class myClass
{
  private static mySpecialObject myObj;
  …
  public static mySpecialObject getSpecialObject()
  {
    if (myObj == null)
      createSpecialObject();

    return myObj;
  }

  private static synchronized void createSpecialObject()
  {
    if (myObj != null)
      return;

    //perform complex initialization
    myObj = new mySpecialObject();
    …
  }
  …
}
```

- *Use private fields.*

Making fields private protects them from unsynchronized access. Controlling their access means these fields need to be synchronized only in the class's critical sections when they are being modified.

- *Use a thread-safe wrapper.*

Provide a thread-safe wrapper on objects that are not thread-safe. This is the approach used by the collection interfaces in JDK 1.2. See "PERF-25: Hashtable and Vector" in this section for more information on the use of this approach to minimize synchronization.

- *Use immutable objects.*

An immutable object is one whose state cannot be changed once it is created. Since there is no method that can change the state of any of the object's instance variables once the object is created, there is no need to synchronize on any of the object's methods.

This approach works well for small objects containing simple data types. The disadvantage is that whenever you need a modified object, a new object has to be created. This may result in creating a lot of small and short-lived objects that have to be garbage collected. One alternative when using an immutable object is to also create a mutable wrapper similar to the thread-safe wrapper.

An example is the String and StringBuffer class in Java. The String class is immutable while its companion class StringBuffer is not. This is part of the reason why many Java performance books recommend using StringBuffer instead of String concatenation. See "PERF-28: StringBuffer" in this section for further discussion on String and StringBuffer.

- *Do not over synchronize.*

Some Java objects (such as Hashtable, Vector, and StringBuffer) already have synchronization built into many of their APIs. They may not require additional synchronization.

- *Do not under synchronize.*

Some Java variables and operations are not atomic. If these variables or operations can be used by multiple threads, you must use synchronization to prevent data corruption. For example:

- Java types long and double are comprised of eight bytes; any access to these fields must be synchronized.

- Operations such as ++ and –- must be synchronized because they represent a read and a write, not an atomic operation.

PERF-25: Hashtable and Vector

Replace Hashtable and Vector with Hashmap, ArrayList or LinkedList if possible.

The Hashtable and Vector classes in Java are very powerful, because they provide rich functions. Unfortunately, they can also be easily misused. Since these classes are heavily synchronized even for read operations, they can present some challenging problems in performance tuning.

One customer asked us for assistance to evaluate the performance and memory leaks in one of its applications. After some instrumentation and analysis of the application, we found they used 45 Hashtables and 215 Vectors to format a very simple HTML page. For a more complicated page, we found they used over 2,000 Hashtables and 6,500 Vectors.

The Hash, Set, and Map interfaces in JDK 1.2 provide both a non thread-safe implementation and a thread-safe wrapper, so the user can have more control over synchronization. Many of the resources listed in the "References and Resources" section explain each of these interfaces in great detail. In summary:

- *Use an Array instead of an ArrayList if the size can be fixed.*

If you can determine the number of elements, use an Array instead of an ArrayList, because it is much faster. An Array also provides type checking, so there is no need to cast the result when looking up an object.

- *Use an ArrayList or LinkedList to hold a list of objects in a particular sequence.*

A List holds a sequence of objects in a particular order based on some numerical indexes. It will be automatically resized. In general, use an ArrayList if there are many random accesses. Use a LinkedList if there are many insertions and deletions in the middle of the list.

- *Use HashMap or TreeMap to hold associated pairs of objects.*

A Map is an associated array which associates any one object with another object. Use a HashMap if the objects do not need to be stored in sorted order. Use TreeMap if the objects are to be in sorted order. Since a TreeMap has to keep the objects in order, it is usually slower than a HashMap.

- *Replace Hashtable, Vector, and Stack.*

Replace a Vector with an ArrayList or a LinkedList.

Replace a Stack with a LinkedList.

Replace a Hashtable with a HashMap or a TreeMap.

Vector, Stack, and Hashtable are synchronized views of List and Map. You can create the equivalent of a Hashtable using:

```
private Map  hashtable = Collections.synchronizedMap (new HashMap());
```

However, bear in mind that even though methods in these synchronized views are thread-safe, iterations through these views are not safe. Therefore, they must be protected by a synchronized block.

- *Avoid using String as the hash key (if using JDK prior to 1.2.2).*

In Java's HashMap or TreeMap implementation, the `hashCode()` method on the key is invoked every time the key is accessed. If the **hash** key is a String, each access to the key will invoke the `hashCode()` and the `equals()` methods in the String class. Prior to JDK release 1.2.2, the `hashcode()` method in the String class does not cache the integer value of the String in an int variable; it has to scan each character in the String object each time. Such operations can be very expensive. In fact, the longer the length of the String, the slower is the `hashCode()` method.

PERF-26: Reuse Objects

Reuse objects instead of creating new ones if possible.

Object creation is an expensive operation in Java, with impacts on both performance and memory consumption. The cost varies depending on the amount of initialization needed when the object is created. Here are ways to minimize excess object-creation and garbage-collection overhead:

- *Use a pool to share resource objects.*

Examples of resource objects are threads, JDBC connections, sockets, and complex user-defined objects. They are expensive to create, and pooling them reduces the overhead of repetitively creating and destroying them. On the down side, using a pool means you must implement the code to manage it and pay the overhead of synchronization when you get or remove objects from the pool. But the overall performance gain from using a pool to manage expensive resource objects outweighs the overhead.

Be cautious when implementing a resource pool. We have often seen the following mistakes in customers' pool management:

- A resource object which should be used only serially is given to more than one user at the same time.
- Objects returned to the pool are not properly accounted for and are therefore not reused, wasting resources and causing a memory leak.
- Elements or object references kept in the pool are not reset or cleaned up properly before being given to the next user.

These mistakes can have severe consequences, including data corruption, memory leaks, a race condition, or even a security problem. Our advice in managing your pool is: keep your algorithm simple.

The "Servlets and JavaServer Pages: Design and Implementation" section includes examples showing how you can use Oracle's built-in JDBC connection caching and the servlet's SingleThreadModel to help manage a shared pool without implementing it yourself.

- *Recycle objects.*

Recycling objects is similar to creating an object pool. But there is no need to manage it, because the pool has only one object. This approach is most useful for relatively large container objects (such as Vector or Hashtable) that you want to use for holding temporary data. Reusing these objects instead of creating new ones each time can avoid memory allocation and reduce garbage collection.

Similar to using a pool, you must take precautions to clear all the elements in any recycled object before you reuse it to avoid memory leaks. You can use the `clear()` method built in to the collection interfaces. If you are building your object, you should remember to include a `reset()` or `clear()` method if necessary.

- *Use lazy initialization to defer creating an object until you need it.*

If initialization of an object is expensive or if an object is needed only under specific conditions, then defer creating it until it is needed.

```
public class myClass
{
  private mySpecialObject myObj;
  …
  public mySpecialObject getSpecialObject()
  {
    if (myObj == null)
      myObj = new mySpecialObject();
```

```
    return myObj;
  }
  …
}
```

PERF-27: Unused Objects and Operations

Avoid creating objects or performing operations that may not be used.

This mistake occurs most commonly in tracing or logging code that has a flag to turn the operation on or off during runtime. Some of this code goes to great lengths creating and formatting output without checking the flag first, creating many objects that are never used when the flag is off. This mistake can be quite expensive, because tracing and logging usually involve many String objects and operations to translate the message or even access to the database to retrieve the full text of the message. Large numbers of debug or trace statements in the code make matters worse.

PERF-28: StringBuffer

Use StringBuffer instead of String concatenation.

The String class is the most commonly used class in Java. Especially in Web applications, it is used extensively to generate and format HTML content.

String is designed to be immutable; in order to modify a String, you have to create a new String object. So String concatenation can result in creating many intermediate String objects before the final String can be constructed. StringBuffer is the mutable companion class of String; it allows you to modify the String. Therefore, StringBuffer is generally more efficient than String when concatenation is needed.

- *Use StringBuffer instead of String concatenation if you repeatedly append to a String in multiple statements.*

Using the "+=" operations on a String repeatedly is expensive. For example:

```
String s = new String();
  [do some work …]
s += s1;
  [do some more work …]
s += s2;
```

Replace the above String concatenation with a StringBuffer.

```
StringBuffer strbuf = new StringBuffer();
  [do some work …]
strbuf.append(s1);
  [so some more work …]
strbuf.append(s2);
String s = strbuf.toString();
```

- *Use either String or StringBuffer if the concatenation is within one statement.*

String and StringBuffer perform the same in some cases, so you do not need to use StringBuffer directly. Optimization is done automatically by the compiler, as illustrated in Table 2–1. In these cases, there is no need to use StringBuffer directly.

Table 2–1 Java2 Compiler Conversion of String and StringBuffer

Before Compiling	After Compiling
`String s = "a" + "b" + "c";`	`String s = "abc";`
`String s = s1 + s2;`	`String s = (new StringBuffer()).append(s1).append(s2).toString();`

- *Use StringBuffer instead of String concatenation if you know the size of the String.*

The default character buffer for StringBuffer is 16. When the buffer is full, a new one has to be re-allocated (usually at twice the size of the original one). The old buffer will be released after the content is copied to the new one. This constant reallocation can be avoided if the StringBuffer is created with a buffer size that is big enough to hold the String. This:

```
String s = (new StringBuffer(1024)).append(s1).append(s2).toString();
```

will be faster than

```
String s = s1 + s2;
```

Oracle HTTP Server Tuning

The Oracle HTTP Server is an Apache-based Server. You can find much of this tuning information for the Apache server in “Chapter 4: Optimizing HTTP Server Performance" in the *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation or from other Apache resources (see the "References and Resources" section).

We are simply listing some of the directives that have a performance impact and that you should examine and tune on your Web server.

PERF-29: TCP/IP Parameters

Tune TCP/IP parameters.

Setting TCP/IP parameters can improve Apache server performance dramatically. We have listed the TCP/IP settings that we recommend for Solaris, along with a detailed explanation for each of these parameters, in the *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation.

PERF-30: KeepAlive

Tune KeepAlive directives.

The KeepAlive, KeepAliveTimeout and MaxKeepAliveRequests directives are used to control persistent connections. Persistent connections are supported in HTTP1.1 to allow a client to send multiple sequential requests through the same connection.

Setting KeepAlive to “On" allows Apache to keep the connection open for that client when the client requests it. This can improve performance, because the connection has to be set up only once. The tradeoff is that the HTTPD server process cannot be used to service other requests until either the client disconnects, the connection times out (controlled by the KeepAliveTimeout directive), or the MaxKeepAliveRequests value has been reached.

You can change these KeepAlive parameters to meet your specific application needs, but you should not set the MaxKeepAliveRequests to 0. A value of 0 in this directive means there is no limit. The connection will be closed only when the client disconnects or times out.

You may also consider setting KeepAlive to “Off" if your application has a large population of clients who make infrequent requests.

PERF-31: MaxClients

Tune MaxClients directive.

The MaxClients directive controls the maximum number of clients who can connect to the server simultaneously. This value is set to 256 by default. You can configure this parameter to a maximum of 1024 if you are using Oracle9*i* Application Server release 1.0.2 and later.

If your requests have a short response time, you may be able to improve performance by setting MaxClients to a lower value than 256. However, when this value is reached, no additional processes will be created, causing other requests to fail. In general, increasing the value of MaxClients does not improve performance when the system is saturated.

If you are using persistent connections, you may require more concurrent HTTPD server processes, and you may need to set the MaxClients directive to a higher value. You should tune this directive according to the KeepAlive parameters.

PERF-32: DNS Lookup

Avoid any DNS lookup.

Any DNS lookup can affect Apache performance. The HostNameLookups directive in Apache informs Apache whether it should log information based on the IP address (if the directive is set to “Off"), or look up the hostname associated with the IP address of each request in the DNS system on the Internet (if the directive is set to “On").

We found that performance degraded by a minimum of about 3% in our tests with HostNameLookups set to "On". Depending on the server load and the network connectivity to your DNS server, the performance cost of the DNS lookup could be much higher. Unless you really need to have host names in your logs in real time, it is best to log IP addresses and resolve IP addresses to host names offline.

PERF-33: Access Logging

Turn off access logging if you do not need to keep an access log.

It is generally useful to have access logs for your Web server, both for load tracking and for the detection of security violations. However, if you find that you do not need these data, you should turn access logging off and reduce the overhead of writing data to this log file.

PERF-34: SSLSessionCacheTimeout

Tune the SSLSessionCacheTimeout directive if you are using SSL.

The Apache server in Oracle9*i*AS caches a client's SSL session information by default. With session caching, only the first connection to the server incurs high latency.

In a simple test to connect and disconnect to an SSL-enabled server, the elapsed time for 5 connections was 11.4 seconds without SSL session caching as opposed to 1.9 seconds when session caching was enabled.

The default SSLSessionCacheTimeout is 300 seconds. Note that the duration of an SSL session is unrelated to the use of HTTP persistent connections. You can change the SSLSessionCacheTimeout directive in the `httpd.conf` file to meet your application needs.

PERF-35: FollowSymLinks

Use FollowSymLinks and not SymLinksIfOwnerMatch.

The FollowSymLinks and SymLinksIfOwnerMatch options are used by Apache to determine if it should follow symbolic links. If the SymLinksIfOwnerMatch option is used, Apache will check the symbolic link and make sure the ownership is the same as that of the server.

PERF-36: AllowOverride

Set AllowOverride to None.

If the AllowOverride directive is not set to None, Apache will check for directives in the `.htaccess` files at each directory level until the requested resource is found for each URL request. This can be extremely expensive.

PERF-37: DMS and Apache Server Status

Use DMS and Apache server status to monitor the performance of the Oracle HTTP Server.

See "Dynamic Monitoring Service (DMS)" on page 2-34 for more information.

Performance Testing

Performance testing is central to performance analysis and evaluation. Here are some general tips. Many of them are self-explanatory and are included in this chapter just as a reminder.

PERF-38: Performance Methodology

Understand the performance methodology.

Terms such as throughput, response time, latency, concurrency, and think time are commonly used in performance analysis. You should understand these terms, how they are used, and how they should be interpreted when running performance tests.

> **See Also:** "Performance Overview" in the *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation

PERF-39: Performance Goals

Determine your performance goals and set realistic performance expectations.

The performance of an application usually varies depending on the load of the system. Set realistic goals for performance at various load levels of the system.

> **See Also:** "Performance Overview" in the *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation

PERF-40: Performance Evaluation

Perform incremental performance evaluation during the development cycle.

Do not wait until the end of the project to do a test cycle. Performance tests should be run regularly after each stage of development. New performance test suites should be added as new features or functions are implemented. If possible, run performance regression tests regularly to compare performance results.

PERF-41: Performance Testing

Run your performance tests on systems that simulate your production environment.

Developers commonly run their functional tests in single-user mode on their workstations or their desktops, which usually have only one CPU. This setup rarely represents the production environment, and it is not adequate for running performance tests. If the application is intended to be used by a large number of concurrent users running on multiple processors, simulate the production environment with a representative workload to study the performance impact.

PERF-42: Test Driver

Understand how to configure your test driver and analyze the result.

Commercial drivers used to simulate HTTP requests can be very effective, but they are often complicated to configure. We have encountered situations where customers set up their driver incorrectly, did not know how to interpret the results, ended up drawing the wrong conclusions, and wasted valuable time in locating the real problems.

PERF-43: Testing Personnel

Assign someone who is experienced in running and analyzing performance tests.

Running performance tests is not a push-button job that can be delegated to an inexperienced engineer. It requires someone who has knowledge and experience with operating systems and databases and who understands the application that is to be analyzed.

Dynamic Monitoring Service (DMS)

Dynamic Monitoring Service (DMS) is a feature introduced in Oracle9*i*AS release 1.0.2.1. It has a set of built-in metrics for the Oracle HTTP Server and the JServ process. For a detailed description of DMS, including an explanation of the metrics that it provides and how you can use DMS to evaluate and monitor the performance of your system, see *Using DMS with Oracle9iAS 1.0.2.2*.

The following are samples of some of these built-in metrics. In Oracle9*i*AS release 1.0.2.1 and later you can display the statistics on a browser or save them in a file to review at a later time. In a future release of Oracle9*i*AS, Oracle Enterprise Manager will provide a comprehensive graphical user interface displaying these metrics in a more interactive form.

Figure 2–2 Global Metrics for the HTTP Server

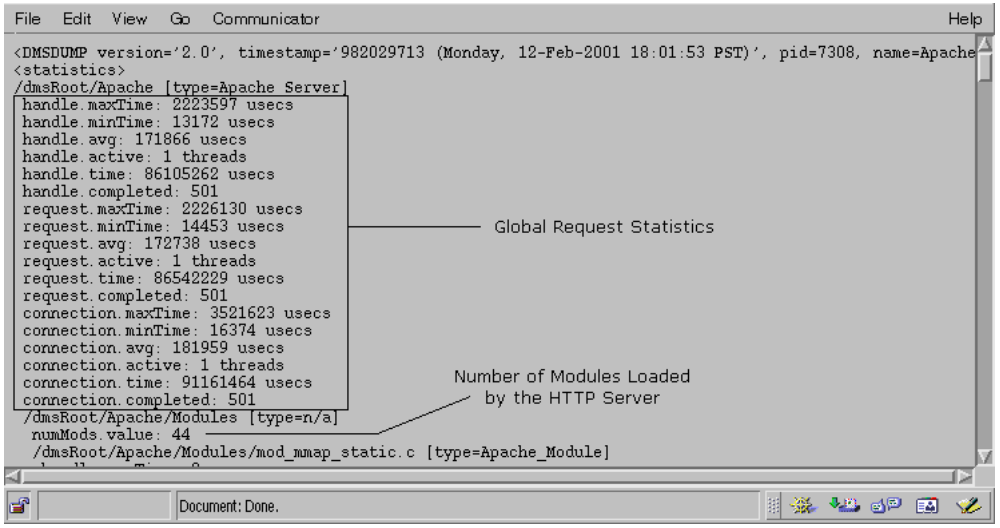


Figure 2–3 Metrics for Each Module Loaded by HTTP Server

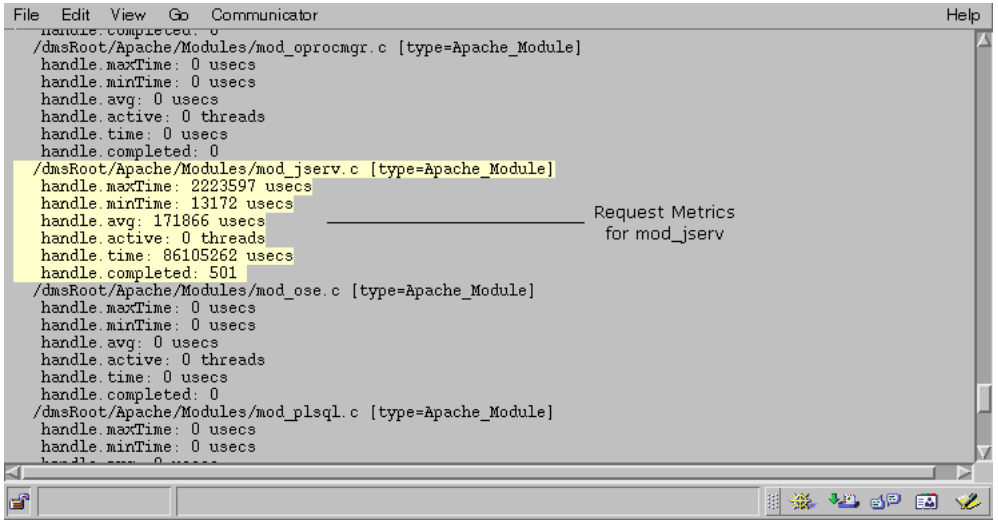


Figure 2–4 Global Metrics for a JServ Process

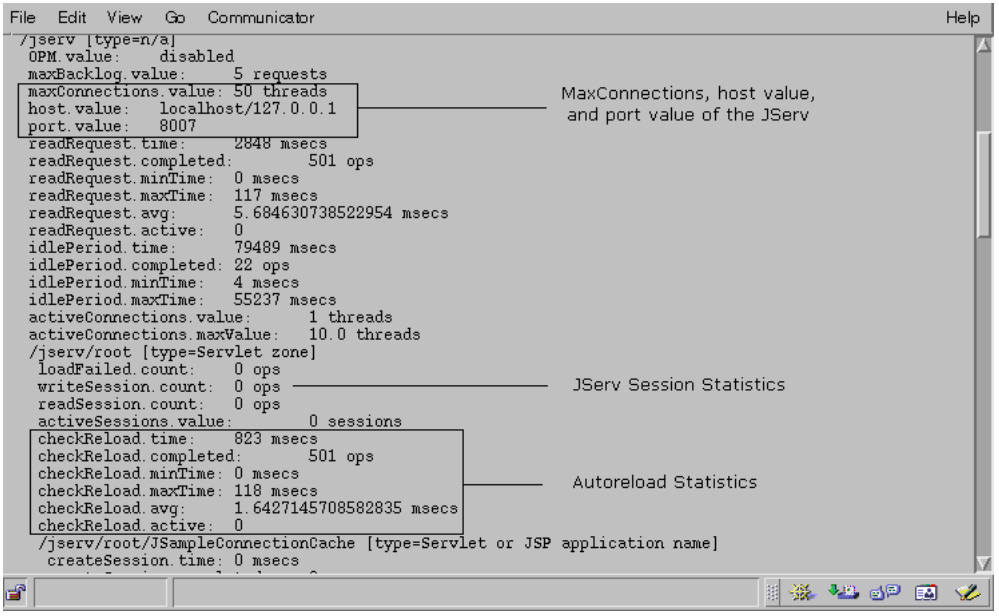


Figure 2–5 JVM Metrics

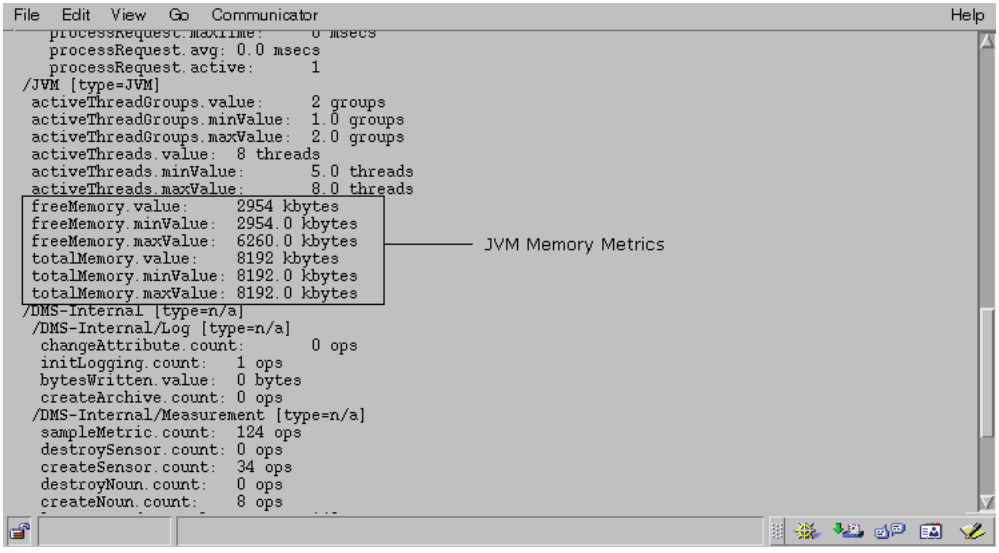
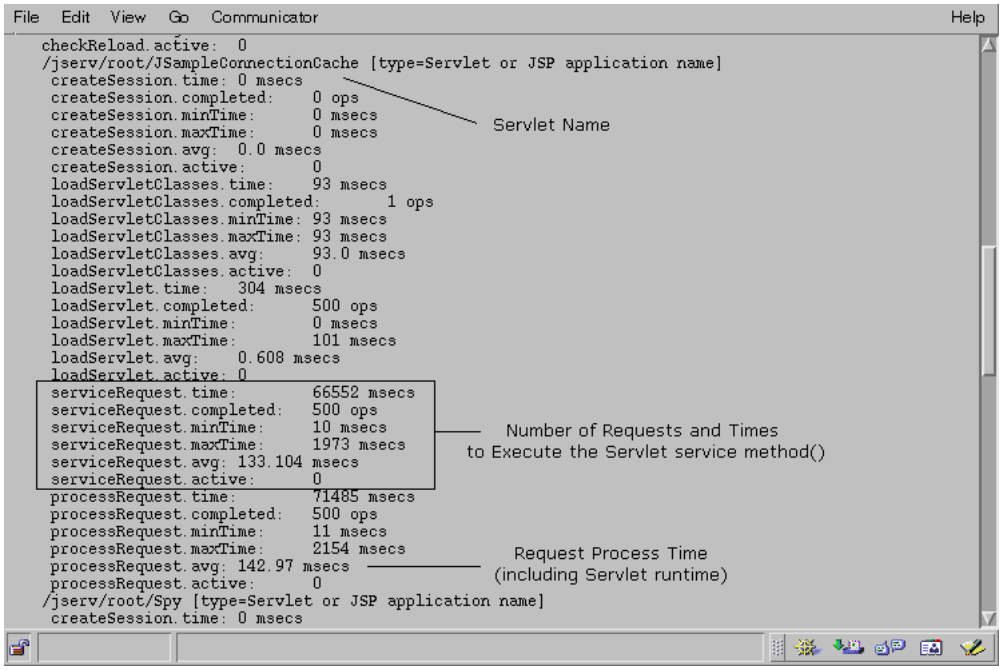


Figure 2–6 Metrics for an Individual Servlet



Code Example

A sample servlet using connection caching and statement caching.

```
import java.io.*;
import java.lang.*;
import java.text.*;
import java.util.*;

import java.sql.*;
import javax.sql.*;
import oracle.jdbc.driver.*;
import oracle.jdbc.pool.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class JSampleConnectionCache extends HttpServlet
{
```

```
// Constants
private final static String DBSERVER_PARAM = "DBServer";
private final static String MAXCONN_PARAM  = "MaxConnections";
private final static String USERID         = "userid";
private final static String GET_STMT       =
  "begin GET_PERFSAMPLE_DATA(?,?,?,?,?,?,?,?); end;";
private final static String RC_NO_ERROR    = "0";
private final static String RC_NOT_FOUND   = "100";
private final static String HTML_TITLE     =
  "<head><title> JSampleConnectionCache </title></head>";

private static OracleConnectionCacheImpl connCache = null;


//---------------------------------------------------------------------------
//
// Servlet life cycle methods
//
//---------------------------------------------------------------------------
public void init(ServletConfig config) throws ServletException
{
  // Get configuration information.
  String dbServer = config.getInitParameter(DBSERVER_PARAM);
  if (dbServer == null)
  {
    throw new ServletException("getInitParameter Exception: " +
       DBSERVER_PARAM + dbServer);
  }

  int maxConn = 0;
  try
  {
    maxConn = Integer.parseInt(config.getInitParameter(MAXCONN_PARAM));
  }
  catch (Exception e)
  {
    throw new ServletException("getInitParameter Exception: " +
       MAXCONN_PARAM + maxConn);
  }

  // Set up a connection cache.
  try
  {
    setupConnectionCache(dbServer, maxConn);
  }
```

```
  catch (Throwable e)
  {
    throw new ServletException(e.toString());
  }
}


public void destroy()
{
  // Close the connection cache created by this instance.
  try
  {
    if (connCache != null)
      connCache.close();
  }
  catch (SQLException e)
  {
  }
}

//---------------------------------------------------------------------------
//
// doGet() is the servlet's main entry for every request.
//
//---------------------------------------------------------------------------
public void doGet (HttpServletRequest httpRequest,
                   HttpServletResponse httpResponse)
throws ServletException, IOException
{
  long execTime = 0;
  Connection dbConn = null;
  CallableStatement dbStmt = null;

  try
  {
    String userName = null;
    String employer = null;
    String allowances = null;
    String addWithholding = null;
    PrintWriter httpOut;

    // Initialize HTTP response header.
    httpResponse.setContentType("text/html");
    httpOut = httpResponse.getWriter();
    httpOut.println("<html>");
```

```
httpOut.println(HTML_TITLE);

// Get user id parameter.
String userid = httpRequest.getParameter(USERID);
if (userid == null)
{
  pgError(httpOut, "Unable to retrieve user information, " +
    USERID + " is null");
  return;
}


// Get data from database.
try
{
  // Get a connection from the connection cache.
  dbConn = connCache.getConnection();

  // Create the prepare statement.
  dbStmt =
    dbConn.prepareCall(GET_STMT);

  if (((OracleCallableStatement)dbStmt).creationState() ==
       OracleStatement.NEW)
  {
    dbStmt.registerOutParameter (2, Types.VARCHAR);
    dbStmt.registerOutParameter (3, Types.VARCHAR);
    dbStmt.registerOutParameter (4, Types.VARCHAR);
    dbStmt.registerOutParameter (5, Types.VARCHAR);
    dbStmt.registerOutParameter (6, Types.VARCHAR);
    dbStmt.registerOutParameter (7, Types.VARCHAR);
    dbStmt.registerOutParameter (8, Types.VARCHAR);
  }
  dbStmt.setString (1, userid);
  dbStmt.execute();

  String retCode = dbStmt.getString(8).trim();
  if (retCode.equals(RC_NO_ERROR))
  {
    String lastName   = (dbStmt.getString(2)).trim();
    String firstName  = (dbStmt.getString(3)).trim();
    String middleName = (dbStmt.getString(4)).trim();
    employer          = (dbStmt.getString(5)).trim();
    allowances        = (dbStmt.getString(6)).trim();
    addWithholding    = (dbStmt.getString(7)).trim();
```

```
          userName = firstName + " " + middleName + " " + lastName;
          pgOut(httpOut, userid, userName, employer, allowances, addWithholding);
        }
        else if (retCode.equals(RC_NOT_FOUND))
        {
          pgError(httpOut, "Record not found for " + userid);
        }
        else
        {
          pgError(httpOut, "Unable to retrieve record for " + userid +
                           " (SQLERROR " + retCode);
        }
      }
      catch (SQLException se)
      {
        throw new ServletException(se.toString());
      }
      catch (Throwable e)
      {
        throw new ServletException(e.toString());
      }
    }
    finally
    {
      try
      {
        // Release the statement object.
        if (dbStmt != null)
          dbStmt.close();

        // Release the logical connection to the connection cache.
        if (dbConn != null)
          dbConn.close();
      }
      catch(SQLException s)
      {
        throw new ServletException(s.toString());
      }

    }
  } // doGet


  //---------------------------------------------------------------------------
  // Servlet private methods
```

```
//---------------------------------------------------------------------------

//
// Setup database connection cache.
private void setupConnectionCache(String dbServer, int maxConn)
throws Throwable
{
  try
  {
    OracleConnectionPoolDataSource connDataSource =
      new OracleConnectionPoolDataSource();

    connDataSource.setURL("jdbc:oracle:thin:@"+dbServer);
    connDataSource.setUser("scott");
    connDataSource.setPassword("tiger");
    connCache = new OracleConnectionCacheImpl(connDataSource);

    // Set maximum number of pooled connections for this cache instance.
    connCache.setMaxLimit(maxConn);

    // Set cache scheme.
 connCache.setCacheScheme(OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);

    // Set statement cache size.
    connCache.setStmtCacheSize(1);

  }
  catch (Throwable e)
  {
    throw e;
  }
}


//
// Return the HTML page.
private void pgOut (PrintWriter httpOut, String userid,
  String userName, String employer, String allowances,
  String addWithholding)
{
  httpOut.println("<body>");
  httpOut.println("<h1>User: <font color=blue>" +
                  userid + "</font> </h1>");
  httpOut.println("<h1>Name: <font color=blue>" +
                  userName + "</font> </h1>");
```

```
    httpOut.println("<h1>Employer: <font color=blue>" +
                    employer + "</font> </h1>");
    httpOut.println("<h1>Number of Allowances: <font color=blue>" +
                    allowances + "</font> </h1>");
    httpOut.println("<h1>Additional Amount of Withholding: <font color=blue>" +
                    addWithholding + "</font> </h1>");
    httpOut.println("</body>");
  } // pgOut


  //
  // Print HTML error page.
  private void pgError (PrintWriter httpOut, String errmsg)
  {
    httpOut.println("<body>");
    httpOut.println("<h1> <font color=red> ERROR: " + errmsg +
                   "</font> </h1>");
    httpOut.println("<hr>");
    httpOut.println("</body>");
  } // pgError
} // JSampleConnectionCache
```

References and Resources

- Hans Bergsten. *JavaServer Pages*. O'Reilly, 2000
- Dov Bulka. *Java Performance and Scalability Volume 1: Server-Side Programming Techniques*. Addison-Wesley, 2000
- Bruce Eckel. *Thinking in Java*. Prentice-Hall, 2000
- Dale Gaudet. *Apache Performance Notes*. http://www.apache.org/docs/misc/perf-tuning.html
- Marty Hal. *Core Servlets and JavaServer Pages*. Sun Microsystems Press/Prentice-Hall, 2000
- Craig Larman, Rhett Guthrie. *Java 2 Performance and Idiom Guide*. Prentice-Hall, 2000
- Jack Shirazi. *Java Performance Tuning*. O'Reilly, 2000
- Bill Venners. *Design Techniques: Articles about Java Program Design*. http://artima.com/designtechniques/index.html
- Peter Wainwright. *Professional Apache*. Wrox Press, 1999.

Oracle Documentation

- *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation
- *Oracle JavaServer Pages Developer's Guide and Reference at* `http://otn.oracle.com/docs/index.htm`
- *Oracle8i JDBC Developer's Guide and Reference* in the Oracle Application Server Documentation Library

3

Pooling

Pooling mechanisms increase application scalability by allowing many users to access the same application without an exorbitant tax on server resources.

This chapter contains these topics:

- Pooling Overview
- Apache Demons
- JDBC Connections
- BC4J Application Modules
- Servlets and Jserv threads
- JavaServer Pages (JSPs)
- Perl
- Portal or PL/SQL
- Database Shared Servers
- Pooling Your Own Resources

Pooling Overview

This section gives an overview of the different pooling mechanisms provided with Oracle9*i*AS:

- Connection pools
- Servlet pools
- Serial reuse

Most Web applications have a much higher number of *browsing* than *active* users. Browsing users have accessed a Web page and are either looking at it or working in it (for example, filling in form elements). Active users have clicked on one of the links in the Web page and are waiting for a reply to come back. We often say that these are *in a call* or *submitted a request*.

Users spend much more time looking at or working in a page than waiting for the next page to come up, at least under normal load. *Pooling* takes advantage of this difference to reduce the amount of server-side resources necessary to serve a large user population. This reduction in necessary resources keeps servers in a well-behaved response zone and avoids sharp degradation of response time due to swapping, network contentions, or other server overheads.

A pool is a collection of instances of a resource—such as a database connection, Java object, or server process—that see **serial reuse** across clients. Instead of having one instance of the resource per client, you create a pool of a small number of instances. You then have each client borrow an instance from the pool when needed and return it when finished.

The resource is typically borrowed at the beginning of a call and returned at the end. Different clients sequentially use the same instance of the resource at different times, but two clients never use the same resource at the same time. When a resource is borrowed, the borrower gets exclusive use of it.

Pooling allows you to:

- Use fewer instances than you serve clients for good scalability
- Amortize resource overhead across multiple clients, improving performance
- Control the load on your servers for better **throttling**

Pooling Example

The concept and benefits are better understood with an example. At any given time a typical e-commerce application may have 10,000 browsing users. Suppose each user spends 10 seconds looking at or working in a page, and each request for the next page takes 100ms on the server side. So each user is active only one percent of the time, or looked at the other way, only one percent of the 10,000 browsing users (= 100 users) are concurrently active. If each request has to go to the database, then only 100 database connections / sessions are needed to service the 10,000 users.

From the server perspective, this is much better than using one connection / session for each user, in which case the database would have to support 10,000 of them at the same time. It is also better than creating and closing a connection at each call, in which case the database would be dominated by the overhead of creating and destroying these sessions. The pool of 100 connections is enough to support 10,000 users, and at the same time it leaves resources available on the server to do other things. Everyone wins.

Pooling versus Sharing

Pooling is different from sharing. Multiple clients use shared resources at the same time; pooled resources are used by only *one* client at a time. Shared resources lower server usage but can add contention points. Pooled resources lower server usage but add queuing points. Consider the following two examples:

- The Java VMs used to run servlets are *shared* across all clients.

  Multiple client threads run at the same time in the Java VM. The clients can impact each other's performance, for example in the case where one client triggers a long garbage collection that prevents another from allocating new Java objects.

- Oracle shared servers, on the other hand, are *pooled* across clients.

  A client being served by a shared server has full use of the process while the request is executed. If all shared servers are in use, then additional clients will be put in a queue until a server becomes available. (Other mechanisms also automatically start new servers up to a maximum number.)

Pools in Oracle9*i*AS

Oracle9*i*AS provides pools at many levels in the request processing pipeline. At the very front end, Apache uses a pool of processes called HTTPD demons to receive and handle HTTP requests. In the backend, the database (when configured in Microsoft Transaction Server mode) uses a pool of processes called *shared servers* to handle SQL requests.

In between, the Apache module mod_plsql uses pools of database sessions to handle requests, and mod_perl effectively has a pool of Perl interpreters (one per Apache demon).

When running Java programs, you can pool your access mechanisms to the database. If you are using JDBC from servlets or JavaServer Pages (JSPs), then you can use JDBC Connection Caches. If you are using BC4J, then you can use Application Module pools. In addition, you can tune two internal pooling mechanisms used by the servlet engine: threads and servlets pools.

The rest of this chapter provides details on these built-in pools in Oracle9*i*AS, and it provides recommendations for building custom pools for your own expensive resources.

Apache Demons

The main entry point into Oracle9*i*AS is the Apache Web server. Apache version 1.x (Oracle9*i*AS release 1.0.2 is shipping Apache 1.3.12) uses a pool of processes called HTTPD demons to handle the client requests. Apache launches an initial set of processes when it starts up. These processes are used in sequence to handle incoming HTTP requests.

At any given time, each process is handling only one client. When the process is done with that client, it goes back in the queue and waits for a request from another client. If the number of clients waiting exceeds the number of available HTTPD demons, then Apache can automatically create new processes to service them. If the load later goes down, then Apache will kill any extra HTTPD processes.

POOL-1: Client-Request Processes

Control the number of Apache client-request processes.

You can easily control the minimum and maximum numbers of demon processes that Apache uses. Apache starts the minimum number when it comes up. You want to set it to handle your server's normal load.

The maximum number can be used to throttle the Web server, by forcing extra clients to wait a bit instead of saturating the machine with HTTPD processes.

> **See Also:** For more information on configuring the HTTPD demons pool, see the *Oracle HTTP Server Performance Guide* in the Oracle9*i* Application Server Platform-specific Documentation

JDBC Connections

The Oracle JDBC 2.0 driver provides a mechanism called *connection caching* that effectively implements pools of connections to the database. This pooling mechanism follows the JDBC 2.0 standard APIs. A JDBC connection cache is a collection of JDBC connections to database sessions. All the sessions are in the same database and are authenticated as the same user. A connection cache can be used from servlets, JSPs, or any Java code running in the JDK.

POOL-2: JDBC Connection Cache

Set up a JDBC connection cache.

A connection cache is represented by an instance of the OracleConnectionCacheImpl class, provided with the JDBC libraries. Your application will usually instantiate this class on startup, keep a reference to the instance in a well-known place, and use it when connections are needed. Full J2EE applications will prefer to bind connection caches within the JNDI namespace, which will avoid having to create them on startup. In that case, the application would just use JNDI APIs to `lookup()` the connection cache when needed. The JNDI layer will take care of instantiating the connection cache the first time and reuse it on subsequent calls.

POOL-3: Connection Authentication

Authenticate connections on the fly.

All connections managed by a given connection cache use the same username and password when accessing the database. This is one of the features that make the connections reusable across clients. You can provide the username and password when you create the connection cache instance or pass it on the fly each time you ask the cache for a connection. Some developers see a security risk in passing the username and password when creating the cache, because the password is then held in a global variable in the Java VM where the cache was instantiated. Our

examples reflect this concern and pass the information only when requesting a connection.

Servlets and Connection Cache

To use a connection cache from servlets running in the JDK, you first create it in the initialization phase of your application (for example in the `init` method of your servlet) and then hold it in a Java static variable. To borrow a connection from the cache, you call one of the `getConnection()` methods of the connection cache. When you are done with the connection, release it by using its `close()` method. This method has been specialized by the connection cache to put the database connection back in the pool instead of terminating it. Another client will be able to borrow that connection as needed.

The following example shows how to initialize a connection cache and make it accessible from a single servlet:

```
public class MyServlet extends HttpServlet{
  // The connection cache is held in a private static attribute of the servlet.
  static private OracleConnectionCacheImpl cache = null;

  // This function initializes the cache if needed and returns it.
  static OracleConnectionCacheImpl getConnectionCache()
    throws SQLException {
    if (cache == null) {
      DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
      cache = new OracleConnectionCacheImpl();
      cache.setURL("jdbc:oracle:thin:@dlsun57:1521:ORCL2");
      cache.setMaxLimit (3);
      cache.setCacheScheme (OracleConnectionCacheImpl.FIXED_WAIT_SCHEME);
    }
    return cache;
  }

  // The doGet method of the servlet is what handles the HTTP requests.
  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println ("<html><body>MyServlet<p>Employee List<ul>");
    Connection conn = null;
    try {
      // Borrow a connection from the cache.
      conn = getConnectionCache().getConnection("scott", "tiger");
```

```
      // Use it as a regular JDBC Connection.
      PreparedStatement stmt = conn.prepareStatement ("select ename from emp");
      ResultSet rset = stmt.executeQuery ();
      while (rset.next()) {
        out.println ("<li>" + rset.getString (1));
      }
      stmt.close ();
      out.println ("</body></html>");

      // This catch block handles SQL exceptions.
    } catch (SQLException e) {
      …
      // The finally block ensures that the connection is returned back to the
      // cache even if an exception is raised.
    } finally {
      if (conn != null) {
        // The close() method returns the connection back to the cache.
        try { conn.close (); } catch (SQLException e) {}
      }
    }
  }
}
```

Most often, your application uses multiple servlets, and you should share a global connection cache across them all. To achieve this, wrap the connection cache within a singleton class accessed by all servlets, as shown in the following example:

```
// Separate singleton class to manage the cache
public class MyCache {
  static private OracleConnectionCacheImpl cache = null;

  public static OracleConnectionCacheImpl getConnectionCache()
    throws SQLException {
    if (cache == null) {
      DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
      cache = new OracleConnectionCacheImpl();
      cache.setURL("jdbc:oracle:thin:@dlsun57:1521:ORCL2");
      cache.setMaxLimit (3);
      cache.setCacheScheme (OracleConnectionCacheImpl.FIXED_WAIT_SCHEME);
    }
    return cache;
  }
}
```

```
// Any servlet of the application can borrow a connection from the cache.
public class MyServlet2 extends HttpServlet
{
  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    …
    try {
      // Get a connection from the cache
      conn = MyCache.getConnectionCache().getConnection("scott", "tiger");
      …
    } finally {
      if (conn != null) {
         try { conn.close (); } catch (SQLException e) {}
      }
    }
  }
}
```

POOL-4: Database Updates

Commit or roll back database updates before returning a connection to the cache.

We recommend that you borrow a JDBC connection at the beginning of a call and return it at the end of the call. You need to make sure that you have committed or rolled back all updates to the database before returning the connection to the cache. If you do not, then the next thread that borrows the connection may see the non-committed changes of the first thread, which would be wrong. A best practice is to include a call to the `rollback()` method in the `finally` block that returns the connection to the cache:

```
try {
   // Get a connection from the cache.
   conn = MyCache.getConnectionCache().getConnection("scott", "tiger");
   …
 } finally {
   if (conn != null) {
    // The rollback ensures that all updates are reverted before returning
    // the connection to the cache.
      try { conn.rollback (); } catch (SQLException e) {}
      try { conn.close (); } catch (SQLException e) {}
 }
```

POOL-5: Linked Servlets

Pass connections across linked servlets.

In a Servlets 2.2 compliant container you can call a servlet from another servlet using a `HttpRequestDispatcher` object. If the servlets in the chain all have to access the database, then they should share the same JDBC connection. An easy way to achieve that is to have the first servlet borrow a connection from the cache and store it in an attribute of the `HttpRequest` that gets passed to the other servlets in the chain. Do not forget to release the connection in a `finally` block in your first servlet:

```
try {
      // Get a connection from the cache.
      conn = MyCache.getConnectionCache().getConnection("scott", "tiger");
      // Hold it in the HttpRequest object.
      request.setAttribute ("myConnection", conn);
      …
      // Call otherServlet that will get the connection from the HttpRequest.
      request.getRequestDispatcher("otherServlet").include(request, response);
      …
    } finally {
      if (conn != null) {
       // The rollback ensures that all updates are reverted before returning
       // the connection to the cache.
         try { conn.rollback (); } catch (SQLException e) {}
         try { conn.close (); } catch (SQLException e) {}
    }
```

If you do not always chain the servlets in the same order, then it may be a good idea to check if a connection is already available in the `HttpRequest` before borrowing one from the cache. The borrowing code would then look like the following:

```
boolean borrowed = false;
conn = (Connection)request.getAttribute("myConnection");
  try {
    // Is there a connection in the HttpRequest?
    if (conn == null) {
      conn = MyCache.getConnectionCache().getConnection("scott", "tiger");
      request.setAttribute("myConnection", conn);
      borrowed = true;
    }
    …
  } finally {
    if (conn != null && borrowed) {
```

```
        // Only return the connection if we borrowed it ourselves.
          try { conn.rollback (); } catch (SQLException e) {}
          try { conn.close (); } catch (SQLException e) {}
      }
    }
```

Holding Connections Across Calls

You will not usually hold borrowed connections across calls, because this would defeat the purpose of the cache. Your cache would have to be larger than necessary. In the worst case you would end up with one connection per client, which renders the cache useless.

One example where it would be necessary to hold a connection is a transaction that spans multiple calls. Suppose the first call locks a row with a SELECT FOR UPDATE, and the next call modifies it with an UPDATE statement before committing the transaction. Most applications avoid these situations, but they are still sometimes necessary. You can hold a connection by storing it in the HttpSession associated with your client:

```
// Holding a connection across calls
public class MyServlet3 extends HttpServlet
{
  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    …
    if (<locking-a-row>) {
      // Get a connection from the cache
      HttpSession session = request.getSession (true);
      conn = session.getAttribute ("connection");
      if (conn == null)
        conn = MyCache.getConnectionCache().getConnection("scott", "tiger");
      session.setAttribute ("connection", conn);
      <lock-the-row>
      …
    } finally {
        // The connection is not returned to the cache here
        if (stmt != null) stmt.close ();
    }
    else if (<updating-locked-row>) {
HttpSession session = request.getSession (true);
        conn = session.getAttribute ("connection");
        <update-the-row>
        session.removeAttribute ("connection");
```

```
    } finally {
         // Here we do return the connection to the cache.
         if (conn != null) conn.close();
  }
}
```

If you hold connections in the HttpSession, then you have to ensure that you eventually return the borrowed connections. The most common mistake is to let the session time out without returning the connection. You can avoid this with the HttpSessionBindingListener mechanism, which lets you bind objects in the session and have these objects notified (execute some code) when they get bound or unbound. See the documentation of the HttpSession class for more information.

POOL-6: *min* and *max* Parameters

Size the connection cache with min and max parameters.

If you use a connection cache, then you will have to tune the `min` and `max` parameters, which control the size of the cache. You will also have to choose one of several policies for when you run out of connections in the cache. Most often you will use the `FIXED_WAIT_SCHEME` policy, which causes additional clients to wait when the pool is exhausted.

The `min` parameter controls the warm-up time of your application. The connection cache will create `min` connections when it is constructed. A large `min` will slow down the startup of your application. But after startup the application will be responsive to a high load, because connections will be available immediately.

If `min` is small, then the application will start up faster. But the initial users will experience delays until the connection cache is filled with enough connections. You should set `min` to a low value (one, for example) during development and to a higher value in production.

The `max` parameter provides a throttle on the database load that your application creates. If the database is dedicated to the application, then you will want to configure the cache to just saturate the database when all the connections are used. You will have to do experiments to find the saturation point.

Use a test harness that simulates many clients hitting your application without any think time between requests. That is, each client executes request after request without any delay between them. As the number of clients increases, you will notice degradation in the response time. At some point you will decide that the response time is too high: you will have reached the scalability limit of your system.

The number of clients at this point is your system's maximum number of database connections. If you have only one Java VM, then it is also the `max` size of the connection cache that you can use. But if you deploy your application on multiple Java VMs, then you will have to divide this number by the number of VMs: each VM will have a fraction of the total number of database connections.

Suppose your nominal response time is 100ms, and it degrades with the number of clients to a maximum-tolerable response time of 2s at 200 clients. In this case, you need a total of 200 maximum connections to your database. If you deploy your application on five Java VMs, then you need a connection cache `max` size of 40 in each VM (40x5=200).

If the database is shared with other applications, then you can follow a similar technique to measure the correct `max` value. However, instead of increasing the number of clients until the application reaches its scalability limit, increase the number of clients until the database reaches the load that you choose to allocate for your application. For example, if you decide that your application will use only 50% of the database resources, then the `max` size of your connection cache will be the number of clients when the database usage reaches 50%.

POOL-7: Cache Usage

Monitor cache usage to tune its performance.

The previous best practice explains how to estimate the `min` and `max` parameters. To ensure that your application performs optimally, you will have to monitor the usage of the cache and tune these parameters. You may have decided that a `min` size of ten was appropriate, only to find that you never use more than three connections from the pool. Or you may have set a `max` of 30, only to find that the application becomes too sluggish long before all the connections are used.

To help you tune the cache, you should report information on the cache usage to your servlet log files. You can use the `getActiveSize()` and `getCacheSize()` methods of the connection cache for this. Active size will tell you how many connections are actually *used* at a given time. Cache size will tell you how many connections have been *opened*, which reflects the peak size of your cache.

Set your `min` parameter below the average active size and your `max` parameter somewhere between the active and peak size.

Note: When using Jserv, the messages printed with the servlets logging APIs go into the file named `jserv.log`.

```
public class MyCache {
  …
  public static OracleConnectionCacheImpl
    getConnectionCache(HttpServlet servlet)
    throws SQLException {
    if (ods == null) {
       …
    }
    // This log call will report cache usage information in the servlet engine
    // log file.
    servlet.log ("Connection Cache: " + ods.getCacheSize() + " connections, "
                     + ods.getActiveSize() + " active.");
    return ods;
  }
}
```

Multiple Applications in the Same Java VM

You will often have multiple applications running together in multiple Java VMs. If you want tight control over database resources usage, then you should have one connection cache for each application and Java VM. This enables you to size the caches independently for each application.

Note for Multithreaded Servlets

If you use multithreaded servlets, then you should be careful not to keep references to the borrowed connection in one of the servlet attributes. This would risk having the same connection used by two calls at the same time, which would just not work. You should keep references to borrowed connections only in local variables, or pass them as arguments to the other Java objects called by the servlet code. In general it is easier and wiser to write single-threaded servlets.

```
public class MyServlet extends HttpServlet
{
  // Do not use a servlet attribute to hold the connection.
  Connection myConn;

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    try {
      // The code below will be problematic for multithreaded servlets.
      // Do not do that.
      myConn = getConnectionCache().getConnection("scott", "tiger");
```

POOL-8: SQLJ

Use SQLJ to reduce database access code.

SQLJ is a pre-processor that greatly reduces the amount of code that you write to access the database from Java. SQLJ can easily take advantage of connection caches. You only have to instantiate the SQLJ `DefaultContext` from a connection obtained from the connection cache, as described in the following code sample:

```
import sqlj.runtime.ref.DefaultContext;

public class MyServlet extends HttpServlet
{
  // This SQLJ statement defines an iterator for EMP names.
  #sql public static iterator EmpIter (String ename);

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    …
    // In SQLJ a JDBC connection is represented by a DefaultContext object.
    DefaultContext ctx = null;
    try {
      // The DefaultContext is initialized from a connection borrowed from the
      // cache.
      Connection conn = getConnectionCache().getConnection("scott", "tiger");
      ctx = new DefaultContext(conn);

      EmpIter iter = null;
      #sql [ctx] iter = { select ename from emp };

      while (iter.next()) {
         out.println ("<li>" + iter.ename());
      }
      …
    } catch (SQLException e) {
      …
      // The finally block ensures that the connection is returned to the cache
      // by closing the DefaultContext.
    } finally {
      if (ctx != null) {
           try { ctx.close (); } catch (SQLException e) {}
      }
    }
```

POOL-9: JDBC Statement Cache.

Use JDBC statement cache.

In addition to the connection cache, Oracle JDBC drivers provide a statement cache. This mechanism enables you to hold prepared statements in JDBC connections and avoid parsing the statements at every call. In Oracle8*i* release 8.1.7.1 and in Oracle9*i* this feature can be combined with connections borrowed from a connection cache. (The Oracle8*i* release 8.1.7 JDBC drivers have a bug that prevents caching statements for cached connections.) We recommend that you use the statement cache together with the connection cache.

> **See Also:** "Statement Caching" in the *Oracle8i JDBC Developer's Guide and Reference* in the Oracle Application Server Documentation Library

JDBC OCI Connection Pools

The JDBC OCI driver provides a mechanism called connection pools, which provides a different functionality than the JDBC connection cache. Where the JDBC connection cache mechanism is pooling connection/session pairs, the JDBC OCI mechanism is pooling only network connections; it still uses a large number of database sessions.

If you were to use the JDBC OCI connection pools mechanism in our example of 10,000 users using a pool of 100 database connections, then you would end up using 100 network connections to access 10,000 database sessions. The mechanism enables you to lower the number of network connections, but not the number of database sessions.

Usually, you want to reduce both connections and sessions. Reducing *only* the number of network connections is useful if you need to use one database session for each client but cannot afford the network connection overhead. In other words, the kind of pooling provided by the JDBC OCI connection pools is useful if you want each database session to be authenticated differently.

Oracle provides other mechanisms to reduce network connections (such as the Oracle Connection Manager—CMAN), and you can also reduce the footprint of sessions in the database by configuring the server in shared server mode. The JDBC OCI connection pools mechanism is just another way of achieving the savings provided by the combination of CMAN and shared server mode.

BC4J Application Modules

Some Oracle9*i*AS applications access the database with Oracle Business Components for Java (BC4J) instead of JDBC or SQLJ. With BC4J you can define Java classes that represent the contents of tables or the results of a query, thereby creating an object-oriented view of the database. The BC4J runtime takes care of calling the database through JDBC, generating all the SQL statements necessary, and opening and closing connections as needed. The main BC4J object that represents a connection to the database is called an *application module*. In addition to the database connection, the application module manages a cache of Java objects materialized from database tables.

POOL-10: Application Module Pools

Configure BC4J to use pools of application modules.

When running in a multithreaded server (a servlet engine, for example), the BC4J runtime can be configured to use a pool of application module objects, similar in effect to a JDBC connection cache. You will typically create a pool of application modules in the initialization phase of your program, then borrow and return modules to the pool on call boundaries.

> **See Also:** *Oracle Business Components for Java Developing Business Components* in the Oracle Application Server Documentation Library

The following example shows how a servlet would borrow application modules.

```
public class MyServlet extends HttpServlet
{
  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    …
    // In BC4J a pool is called ApplicationPool.
    ApplicationPool pool = null;
    EmpModuleImpl am = null;

    try {
      // Here we get access to the pool from a singleton class MyCache.
      pool = MyCache.getPool();

         // And we check-out one application module from the pool.
      am = (EmpModuleImpl)pool.checkout();
```

```
      // We use the application module as we would usually do.
      EmpNamesImpl vo = am.getEmpNames ();
      vo.executeQuery();
      while (vo.hasNext ()) {
        EmpNamesRowImpl row = (EmpNamesRowImpl)vo.next ();
        String name = row.getEname();
        out.println ("<li> " + name);
      }

    } catch (Exception e) {
      …
      // The finally block ensures that we return the application module to the
      // pool when we are done.
    } finally {
      if (am != null) pool.checkin (am);
    }
  }
}
```

Just as in the case of the JDBC connection cache, we use a wrapper class to encapsulate the initialization and creation of the pool of application modules.

```
import oracle.jbo.*;
import oracle.jbo.common.ampool.*;

public class MyCache {
  public static ApplicationPool getPool() throws Exception {
    // Ask the BC4J runtime if it knows about the pool.
    ApplicationPool pool = PoolMgr.getInstance().getPool("EmpAMPool");
     if (pool == null) {
      // If not create the pool here.
      Hashtable env = new Hashtable (10);
      env.put (Context.INITIAL_CONTEXT_FACTORY, JboContext.JBO_CONTEXT_FACTORY);
      env.put (JboContext.DEPLOY_PLATFORM, JboContext.PLATFORM_LOCAL);
      PoolMgr.getInstance().createPool("EmpAMPool", "pools.bc4j.emp.EmpModule" ,
                           "jdbc:oracle:thin:scott/tiger@localhost:1521:orcl2",
                                       env);
      pool = PoolMgr.getInstance().getPool("EmpAMPool");
    }
    return pool;
  }
}
```

Servlets and Jserv threads

The servlet engine is a Java program that receives requests from Apache and handles them by invoking servlets. The servlet engine can be configured to use pools of Java threads and pools of Java servlet instances to serve the application better. Configuring these pools is not easy, and the procedure varies depending on whether you are using single-threaded or multithreaded servlets.

By definition, a single-threaded servlet implements the SingleThreadModel interface. A servlet that does not implement this interface is a multithreaded servlet. Single-threaded servlets will be in only one thread (or call) at a time, so the Java instances of a given single-threaded servlet class constitute a pool of resources used by the servlet engine.

The number of instances determines the number of clients that will be able to execute the servlet concurrently. If clients want to execute the servlet, but no instances are available, then they will wait for one of the other clients to be done. You can control the `min` and `max` number of instances used by the servlet engine for single-threaded servlets.

Multithreaded servlets require only one instance. The number of clients executing the servlet at the same time is controlled by the size of the thread pool used by the engine.

The *Oracle HTTP Server Performance Guide*, in the Oracle9*i* Application Server Platform-specific Documentation, explains how to tune the thread pool and the servlet instance pool. In general you can always start with the same size for the two pools and tune from there. Be aware that you will have to measure the performance of your application under different loads to do any tuning at all. Trying to tune without any performance measurement is a waste of time and resources.

POOL-11: Servlet Attributes

Do not pool resources by holding them in servlet attributes.

When you decide to use SingleThreadModel servlets, it is tempting to employ the pool of servlet instances for pooling. The servlet engine will serially reuse instances of your servlet class. So if you hold reusable resources in the servlets attributes, then these resources will automatically be serially reused with the servlets. Typically, you might be tempted to allocate one JDBC connection for each servlet instance, holding it in an attribute of the servlet. As the servlet instances get serially reused across clients, so will the JDBC connections. You can also be tempted to hold prepared statements or other objects that are expensive to create.

You should resist these temptations, however, because this kind of pooling is often shortsighted. As your application grows it will use more and more different servlet classes, and the servlet engine will have pools of instances for each of them. If all these servlets require a JDBC connection, then you will end up using more connections than you would have used with a regular JDBC connection cache.

For example, if you have 10 different servlet classes and configure the servlet engine for 10 instances, then you will end up with 100 JDBC connections. Furthermore the connections will not be passed across servlets or JavaServer Pages as described earlier. For these reasons we do not recommend that you pool resources by holding them in servlet attributes.

```
// Wrongly storing a JDBC connection in a servlet instance attribute

public class MyServlet extends HttpServlet implements SingleThreadModel
{
 // Instance variables
 private Connection dbConn;  // Will wrongly be used to hold connection

 public void init(ServletConfig config) throws ServletException
 {
  // Create a connection and hold it in a private attribute.
  dbConn = DriverManager.getConnection ("jdbc:oracle:oci:…");
 }

 public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
 {
  try {
   // Use the connection from the private attribute.
   PreparedStatement stmt = dbConn.prepareStatement ("select ename from emp");
   …
  } finally {
   // Here you do not close the connection, because you keep it in an attribute.
  }
 }
}
```

JavaServer Pages (JSPs)

Developers often prefer JSPs to servlets, especially for pages that do not have much dynamic content. You can access JDBC connection caches from JSPs by using an *application-scoped* instance of the predefined Oracle ConnBean class. This bean is a wrapper for a JDBC connection cache and will let you easily borrow and return connections from a JSP.

POOL-12: ConnBeans

Use application-scoped ConnBeans with JSPs.

Note that the example provided with the Oracle JSP 1.1 installation is using a session-scoped ConnBean, which defeats the purpose of the cache. If you store the ConnBean in the session scope, then you will have one connection pool for each HttpSession object. This means that the connections will not be reused across clients. To reuse the connections across all clients you must have an application-scoped ConnBean bean.

Here is the correct code sample:

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-- This creates an application scoped connection bean and initializes it. --!>
<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean" scope="application">
<jsp:setProperty name="cbean" property="User" value="scott"/>
<jsp:setProperty name="cbean" property="Password" value="tiger"/>
<jsp:setProperty name="cbean" property="URL"
                 value="jdbc:oracle:thin:@dlsun57:1521:orcl2"/>
<jsp:setProperty name="cbean" property="PreFetch" value="5"/>
<jsp:setProperty name="cbean" property="StmtCacheSize" value="2"/>
</jsp:useBean>

<html><body>
MyJSP<p>
Employee List
<%
    try {
      // Make the connection.
      cbean.connect();
      String sql = "SELECT ename, sal FROM scott.emp ORDER BY ename";
      // Get a cursor bean.
      CursorBean cb = cbean.getCursorBean (CursorBean.PREP_STMT, sql);
      out.println (cb.getResultAsHTMLTable());
      // Close the cursor bean.
```

```
      cb.close();
    } catch (SQLException e) {
      out.println ("<pre>" + e + "</pre>");
      // The finally block returns the connection to the cache by closing the
      // connection bean.
    } finally {
      // Close the bean to close the connection.
      cbean.close();
    }
%>
</body></html>
```

As in the case of servlets, you would likely have several JSPs using the same connection cache. In that case you would not directly initialize the connection bean from within any of the JSPs. You would instead implement your own singleton class as a wrapper of the connection bean, as we did for the servlets examples. Each JSP would borrow connections from that class.

POOL-13: Single-Threaded JSPs

Do not use single-threaded JSPs.

JSPs are compiled to servlets by the JSP compiler, which by default generates multithreaded servlets. At runtime there will be only one Java instance of the servlets, and multiple clients will be executing it in different threads. The number of clients simultaneously executing the JSP is controlled by the thread pool parameters of the servlet engine.

You have the option to generate a single-threaded servlet from a JSP. But the Oracle JSP engine does not manage pools of single-threaded instances; it only ever creates one instance of a single-threaded JSP. Therefore, all clients will be serialized when accessing a single-threaded JSP. So we do not recommend single-threaded JSPs.

This may seem at odds with our recommendation that you use single-threaded servlets. Why do we not also recommend single-threaded JSPs? In general, we prefer single-threaded servlets because it is often difficult to write servlets that function well in a multithreaded environment. With JSPs it is much more difficult to write a non-thread safe page, so it is OK always to use a multithreaded model.

Perl

Perl programs run in memory with the Apache HTTPD demons. This means that there is effectively a pool of Perl interpreters that are serially reused across clients, as the HTTPD demons are.

POOL-14: Perl Programs

Pool database connections from Perl programs.

The global variables of Perl interpreters survive calls and can be used by different clients. You will be able to pool database connections from Perl programs by simply keeping a reference to the database connection in a Perl global variable. You will end up with a pool of the same size as the pool of HTTPD demons.

Portal or PL/SQL

The mod_plsql module delegates the handling of HTTP requests to the backend database. URLs contain the names of PL/SQL programs to execute in response to a request. Module mod_plsql uses one database connection/session for each Apache HTTPD demon. Because the HTTPD demons are serially reused across clients, so are the sessions used by mod_plsql. If you configure Apache with a maximum of 100 HTTPD demons, then you will use a maximum of 100 database connections to execute PL/SQL.

Apache will quickly start new HTTPD demons when the HTTP load goes up, which will result in corresponding database sessions also starting quickly if the requests are handled by mod_plsql. To make the session start up faster and control the load on the backend database, we recommend that you configure it in shared server mode. In shared server mode, the creation of a new session does not require the creation of a new process or thread on the backend server.

Database Shared Servers

So far we have described Oracle9*i*AS pooling support at the Apache, Java, PL/SQL, and Perl levels. One additional important pooling mechanism is the database shared server mode, which provides pools of database server processes that are serially reused across clients. Each client still has a database session and a connection to the server, but the session is only a piece of shared memory instead of a full-fledged server process (on Windows machines server processes are actually OS threads, while on Unix machines they are processes).

POOL-15: Shared Server Mode

Configure the database in shared server mode.

When you configure the database in shared server mode, you can decide to pre-spawn processes and have the database automatically start new processes up to a maximum number. You end up with fewer database processes than in the default dedicated server mode, but you have to allocate more memory to the shared pool in the `init.ora` parameters. Re-using processes across clients increases database scalability and makes session creation cheaper and faster.

Pooling Your Own Resources

So far we have described the built-in pooling mechanisms in Oracle9*i*AS. Advanced programmers may also be interested in implementing pools for the expensive resources used by their applications. For example, if your application accesses a legacy system through some proprietary network protocol, then you may want to define a pool of connections to this system that you will serially reuse across clients. The important points to consider when pooling resources are the following:

- Statelessness

  You should pool resources only if they do not keep per-client residual state after being returned to the pool.

- Reusability

  It is worthwhile to pool your own resources only if they are indeed reusable across different clients. For example, resources that require client-specific authentication (that is, a distinct login for different clients) are not good candidates for pooling.

- Resource cost versus overhead

  A pool is beneficial if the resources are expensive to create or initialize and if the cost can be amortized across many reuses. There will be little benefit from a pool of inexpensive Java objects, such as small footprint Java Beans that perform auxiliary calculations for JSPs. Indeed, the pooling overhead may be more expensive than recreating and dropping these beans at each call. Only measurements of your particular application can tell you if a pool is beneficial or not.

- Pooling versus sharing

  Pooling is intended to provide serial access to a resource, and during access the resource is fully owned by the calling program. This is to be contrasted with sharing, where multiple calling programs or threads access the same resource at the same time. Some objects are better shared than pooled. For example, pricing rules for an e-business site should be a cache of sharable data in memory, rather than a pooled resource.

If you decide to pool your own resources, then it is quite easy to write pooling code similar to what the JDBC connection cache provides. If you are accessing the resource from a JSP, then it is a good idea to gain access to the resource from a page-scoped Bean and release the resource at the end of the page.

4

HTTP Security

This chapter details a number of security best practices for the Oracle HTTP Server. It will not address security issues pertaining to access of the Oracle database.

This chapter contains these topics:

- Security Overview
- Firewall Architecture
- Process Development and Deployment
- Global Server ID Certificates and 128-Bit Encryption
- Performance Issues
- Client Certificates
- EXPORT versus DOMESTIC Encryption Issues

Security Overview

Unlike most other system and application areas, security is a very open-ended concern, simply because there is no specification which states how one is *allowed* to break security. Oracle makes no claim, therefore, that this document is a comprehensive review of all Oracle HTTP Server security issues.

Security problems pertaining to intranets are widely understood, because intranets have been deployed for many years. IT managers and personnel have determined important intranet security risks and the appropriate levels of response to combat these risks.

This is not the case for Internet issues, where risks are not generally understood, and appropriate responses to risks have not been generally agreed upon. We will focus, therefore, on issues which matter primarily in Internet environments. We will give only a glance here and there to issues which matter primarily in intranet-restricted applications and services.

We used several criteria when selecting topics for this chapter. To be included, the information should:

- Be unfamiliar to people moving from the intranet world to the Internet
- Reduce exposure for major security problems
- Avoid massive changes to deployment or application design

We expect our recommended best practices will be rewritten on a regular basis, enhanced and modified in response to the needs of the Oracle community. Many of the topics first appeared as questions and issues on Oracle security-related discussion forums, which we feel gives them especially high priority.

Firewall Architecture

This section discusses firewall architecture from the perspective of organizations wishing to provide Internet-accessible services. We will ignore services available only on intranets, browsers attached to the Internet that access services on the Internet, and intranet-attached processes which need access to Internet services.

There is no single best architecture for accommodating Internet requests requiring access to corporate intranets. Instead, trade-offs must be made between two competing goals:

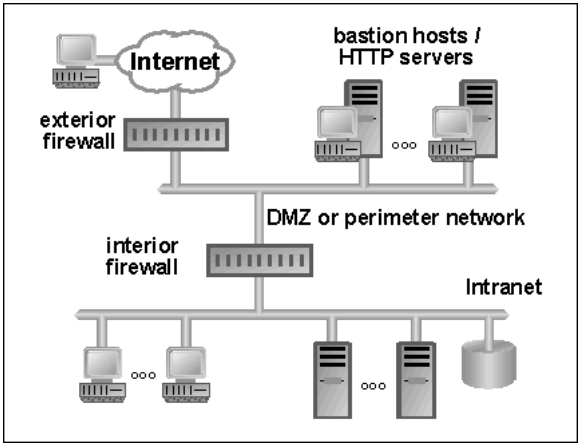
- Security of the intranet against Internet attack
- Ease of access to services by both Internet and intranet clients

Neither goal can be totally met, because complete security means no access to services, while complete ease of access means that anyone is free to peruse, corrupt, or modify corporate sites. We can only try to reach a balance between these goals, whose relative priorities are often unclear.

Oracle recommends two approaches, which we believe should satisfy the requirements of most of our customers:

- DMZ with **bastion hosts**
- Switched connection DMZ hosts

Figure 4–1 Firewalls with Bastion Host

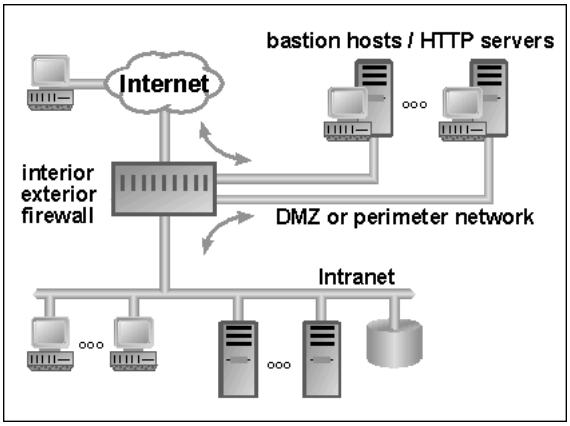


For the purposes of this discussion of Oracle's recommendations, network architecture can be divided into the three regions shown in Figure 4–1. On one side is the wide world of the Internet; on the other side is the corporate intranet. In the middle is the De-Militarized Zone or DMZ (from the military term for an area between two opponents where fighting is prevented), separated from both of them by devices called firewalls. These firewalls block or allow data transfers based on IP address, port, protocol type, or some combination of these. They can also employ **stateful inspection** technology to detect illegal protocol transitions. For more information on stateful inspection, see "SEC-1: Server Placement" on page 4-5.

Firewalls are sometimes defined as including the DMZ, the exterior firewall, and the interior firewall parts of Figure 4–1. In this less common definition, what we call the interior firewall is typically called a router.

The bastion hosts shown in Figure 4–1 are well-fortified servers running initial point of contact protocols, such as HTTP or SMTP. They should be set up with the expectation that outsiders will attempt to break into them. Special care should be taken to ensure that break-ins are difficult. If a break-in does occur, then there should be good fault containment.

Figure 4–2 Switched Connection DMZ Hosts



The switched connection DMZ architecture shown in Figure 4–2 takes advantage of newer firewall technology, which allows inexpensive switched connection attachments of servers. With switched connections one server cannot see the traffic generated by another server, except for broadcasts. This provides a major fault containment benefit compared to bussed connections, where all devices attached to the bus can view traffic to and from other devices on the bus.

While we recommend these approaches, we do not mean to exclude all alternatives. Other reasonable network structures might give higher priority to security or ease of access. Consideration of the issues, risks, and costs of these alternative approaches, however, should precede any deployment decision.

It should be noted that outgoing requests should be handled differently and have different policies than incoming requests. Thus, a reasonable policy might be to allow outgoing FTP requests but no incoming FTP requests.

SEC-1: Server Placement

Place servers providing Internet services behind an exterior firewall of the stateful inspection type.

It is very important that the servers used to provide Internet services be placed in the appropriate part of the service provider's network architecture. In almost no cases should servers be placed directly on the Internet, where they are vulnerable to too many forms of attack. They should instead be behind an exterior firewall.

The best firewall type is the stateful inspection type, such as those available from Checkpoint (`http://www.checkpoint.com`) and other vendors. Stateful inspection means that the firewall keeps track of various sessions by protocol and ensures that illegal protocol transitions are disallowed through the firewall. This blocks the types of intrusion which exploit illegal protocol transitions.

SEC-2: Restrict Traffic Through Exterior Firewall

Set exterior firewall rules to allow Internet-initiated traffic only through specific IP and port addresses where SMTP/POP3/IMAP4 or HTTP services are running.

Generally, it is best to provide only SMTP/POP3/IMAP4 (e-mail) and HTTP (browser) services to the Internet, because other protocols are too vulnerable to attack. Where it is feasible, it is best to provide the HTTP and SMTP/POP3/IMAP services on the DMZ, while running applications or accessing databases on the intranet part of the network.

IP and port combinations which are not assigned to running programs should not be permitted. Once messages are received by the HTTP or SMTP/POP3/IMAP server, they can be forwarded from DMZ processes to intranet processes.

Handling of other protocols by processors attached to the DMZ, especially raw TCP/IP or UDP, is not recommended. FTP use results in major security vulnerabilities, because it essentially sends usercodes and passwords in the clear. IIOP opens too many ports without waiting system processes attached. If handling these protocols on the DMZ is a requirement, then the processors doing the handling should have no incoming access to the intranet.

SEC-3: Restrict Traffic Through Interior Firewall

Set interior firewall rules to allow messages through to the intranet only if they originate from servers residing on the DMZ.

Intranet-initiated requests to the DMZ are not a problem, but no direct access to the intranet from the Internet should be allowed. All incoming messages *must* first be processed on the DMZ.

SEC-4: Proxies on the DMZ

Send outgoing messages through proxies on the DMZ.

Messages originating from processes in the intranet can be passed directly to the Internet. For more security, they can be forwarded to the Internet by *proxies* on the DMZ. Many proxy types exist for the different protocols typically operating over TCP/IP. If even more security is desired, then we recommend the switched connection DMZ host architecture shown in Figure 4–2.

SEC-5: Safeguard Information of Record

Do not store the information of record on bastion hosts.

Information and processing should be segmented such that bastion hosts provide initial protocol server processing and generally do not contain information of a sensitive nature. They should certainly not contain the information of record. That is to say, updates or corruption to information on bastion hosts should not result in updates to the database of record. The database of record and all sensitive processing should reside on the intranet.

SEC-6: Restrict Traffic to Approved Types

Disallow all traffic types unless specifically allowed.

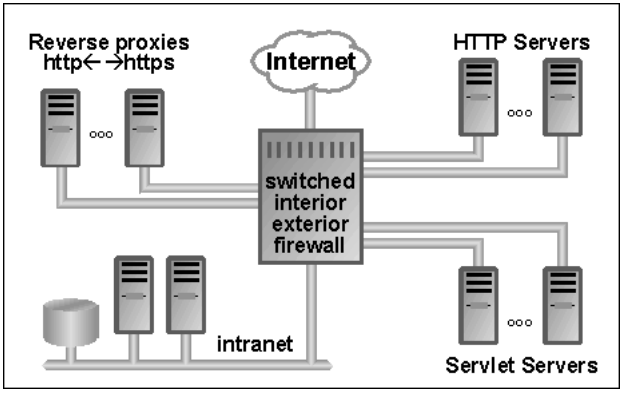
No one can predict what form the next attack on your network might take, and disallowing only the forms taken by past attacks will always leave you one step behind the attackers. We recommend instead that you disallow all types of traffic not required by your organization.

With the bussed connection firewalls discussed in the previous best practices, rules allowing or disallowing traffic between different server/hosts can become quite complex. While in the past there was often concern with building multiple levels of DMZs for better fault containment, most such questions are eliminated with the use of switched connection firewalls.

Note: Some protocols are not encrypted. One example is AJP, the protocol between the Oracle HTTP Server and JServ. When the Oracle HTTP Server and the JServ server are on different computers, the use of switched rather than bussed connections provides significant additional fault containment.

Some of the previous best practices, applicable to DMZs with bastion hosts, apply as well to switched connection DMZ architecture. Routing all incoming traffic to a DMZ server before forwarding it to the intranet is still a good idea. Also still a best practice is banning messages from the Internet to the DMZ, if the messages have IP addresses used in the DMZ. This preserves the DMZ concept, even though the DMZ is no longer a formal LAN or network.

Figure 4–3 Complex Switched Configurations



Switched connection DMZ hosts architecture allows secure segregation of processing tasks. In Figure 4–3 for example, inexpensive servers assisted by cryptographic hardware are used to convert HTTP to HTTPS, while separate hosts are used to segregate HTTP servers from servlet servers on the DMZ.

The switched interior / exterior firewall in this example, which may be built from a number of distinct pieces of hardware and software acquired from different vendors, can provide quite complex routing rules.

Incoming HTTPS traffic might get routed first to reverse proxies, which would convert the HTTPS protocol to HTTP. Such traffic could then be routed to HTTP servers, where requests for static content might be satisfied, while dynamic content

requests might be routed to servlet servers. The servlet servers might then route requests to the intranet for further processing and database access. Another category of processors might provide Web cache capabilities.

This architecture provides excellent fault containment. If one of the HTTP servers is compromised, then it cannot see the traffic from other HTTP servers or servlet servers. Further, rules provided to the firewall could prevent compromised servers accessing other servers on the DMZ or intranet, thereby preventing more corruption and theft. Oracle plans to explore such architectural alternatives for many of Oracle's products soon in a Firewall and Load Balancing white paper.

> **See Also:** For information about stateful firewalls and Checkpoint products, visit `http://www.checkpoint.com/`

Process Development and Deployment

Attacks on corporate networks often attempt to trick components with high levels of privilege into revealing information or modifying internal infrastructure, thereby allowing further incursion into protected resources. This section presents best practices pertaining to the setting of program privileges, including usercode and password management

SEC-7: Privileges and Modules

When assigning privileges to modules, use the lowest levels adequate to perform the module's functions.

This does not prevent trickery, but it limits the damage if a module is taken over or tricked. Faults are better contained.

SEC-8: Buffer Overflow

Ensure that programs are reviewed against buffer overflow for received data.

Buffers can overflow into data structures, resulting in a number of exploitation types—especially denial of service. Buffer overflows are considered by many to be the leading area of security vulnerability and as such deserve special consideration.

SEC-9: Cross-Site Scripting Attacks

Ensure that programs are reviewed against cross-site scripting attacks.

These attacks typically trick HTML and XML processing via input from browsers (or processes which act like browsers) to invoke scripting engines inappropriately. In one of the attack's basic forms, the attacker enters various escape characters such as > or < when presented a form for regular input via a browser. With careful crafting, the attacker can cause a script engine to process the attacker's script using the security level of the script processing engine (which may be quite high).

Consider a simple example. A form is presented to a browser requesting a description of a desired good or service. The attacker enters a bogus description along with > and < escape characters. The escape characters would later cause the output processor to process Javascript (entered as part of the bogus description) when the description was replayed. The script could read protected information from the server and then send it to the attacker via SMTP (e-mail) message.

No simple and effective solutions to this problem exist. Each application writer needs to write code to scan input to ensure that such trickery is not being attempted.

SEC-10: Exported Products

All U.S. built products containing cryptographic software should be reviewed by the building company's export control unit months in advance of production or Beta shipment outside the U.S. or Canada.

A one-time review by the U.S. Department of Commerce is required for all U.S. built products using encryption technology, if the products are to be shipped outside the U.S. and Canada. Legal penalties exist for exporting such products without proper license. This review, which can take many weeks, applies to all new products and may apply to patched or new versions of existing products as well.

In order to meet on-time delivery of products to the international market, it is essential that development consult with the corporate export control department months in advance of international shipment for both product Beta and production use. Corporate export control will advise development staff on the proper processes to ensure that appropriate licenses are obtained in time.

Where cryptographic functions are needed, use already reviewed facilities (that is, common code). This will keep the one-time review as short as possible. When shipping new versions of existing products, avoid changes to portions involving encryption if possible. This may eliminate delays for governmental review entirely.

SEC-11: Passwords and Accounts

When deploying software, change all default passwords and close accounts used for samples and examples.

The risks here should be clear.

SEC-12: Security Patches

Apply all relevant security patches.

Check Metalink (`http://metalink.oracle.com/`) and TechNet (`http://technet.oracle.com/index.html`) for current security alerts. Many of these patches address publicly announced security holes.

SEC-13: Unused Services

Remove unused services from all hosts.

Examples of unused services are FTP, SNMP, NFS, BOOTP, and NEWS. It is almost always worthwhile finding ways to eliminate FTP, because it is especially noxious. HTTP or WebDAV may be a good alternatives.

SEC-14: Root Privileges

Limit the number of people with root privileges.

SEC-15: "r" Commands

Disable the "r" commands, such as rlogon and rsh, if you do not need them.

See Also:

- `http://www.iso.ch` for ISO Common Criteria for security evaluations (ISO 15408)
- `http://csrc.nist.gov/publications/fips/fips1401.htm` for FIPS 140-1 Security Requirements for Cryptographic Modules
- `http://www.cert.org` for information on CERT, a nonprofit organization helping other organizations defend against network attack (good resource for reviewing deployment criteria, and CERT site monitors security alerts)
- `http://www.apache.org/info/css-security/` provides an excellent reference on the cross-site scripting attack problem

Global Server ID Certificates and 128-Bit Encryption

In this section, we explain why the Verisign 40-bit certificates are appropriate for use by nearly all organizations which desire bulk encryption protected by 112-bit, 128-bit, or 168-bit key sizes.

As a direct result of export rules in force until early 2000, Web browsers and server software containing encryption technology were often built in DOMESTIC and EXPORT versions—both official designations of the U.S. Department of Commerce. The DOMESTIC versions used strong encryption, and the EXPORT versions used weak encryption.

Many users outside the United States and Canada (and some domestic users as well) therefore currently have weak-encryption versions of popular browsers like Netscape Navigator and Microsoft Internet Explorer. In this context, weak encryption means key sizes of 64 bits or less (including 40-bit keys) for RC4 and DES bulk encryption. Strong encryption means key sizes greater than 64 bits, including the common 112-bit, 128-bit, and 168-bit key sizes.

Many organizations (especially in the financial, securities, medical, and insurance markets) believe that weak-encryption SSL communication is unacceptable for their applications. They have successfully lobbied for export laws and international agreements to allow single-use licenses for strong-encryption server products in their markets.

Unfortunately, use of strong-encryption server products with weak-encryption browsers results in weak-cryptography SSL sessions. So even with their licensed

strong-encryption servers, these organizations would often be limited to unacceptable weak-encryption SSL sessions when communicating with their customers.

Verisign, the largest supplier of server certificates, responded to this problem by developing a Global Server ID (GSID) Certificate and getting approval for it from the U.S. Department of Commerce. The GSID certificate contains a signed digital right, which will be called the *step-up* digital right in this discussion.

In conjunction with development of the step-up digital right, browser and server logic supporting SSL was revised such that when a GSID certificate was used in a server, weak-encryption (that is, EXPORT version) client browsers would automatically step up their encryption strength from weak encryption to strong encryption.

Global Server IDs are now of limited utility, however, because the strong encryption export ban was lifted early in 2000. Now anyone outside the handful of countries designated as "terrorist" by the U.S. State Department can legally download a strong-encryption browser.

Note: There are different types of X.509 V3 certificates. In describing their X.509 v3 certificate that includes the *step-up* digital right, Verisign uses the terms 128-bit, Global Server ID, Secure Site Pro, and 128-bit global server IDs. Other companies (for example, Baltimore) offer X.509 v3 certificates with the *step-up* digital right and use other names for them. But because Verisign has about 90% of the market, its names are often applied to X.509 V3 certificates from other vendors.

This is unfortunate, because Verisign's naming system is particularly misleading. Certificates without the step-up digital right are now called 40-bit certificates, while certificates with the step-up digital right are called 128-bit certificates. While it is true that the 128-bit certificate will allow 128-bit encryption sessions, it is also true that the 40-bit certificates will allow 128-bit encrypted sessions—if both the browser and the server support 128-bit sessions. Further, X.509 certificates are not actually 128-bit or 40-bit. They do contain a key; but it would likely be 512-bit, 1024-bit, or 2048-bit, and it is not for bulk encryption (which is what the 128-bit and 40-bit key sizes reference).

SEC-16: Fail Weak Encryption

Configure your Web server to fail attempts to use weak encryption. Display an error page explaining the need to upgrade client browsers to 128-bit (strong encryption).

If you are considering the Verisign 128-bit certificate to ensure 128-bit encryption, then use the 40-bit certificate (or an equivalent certificate from another vendor) instead, and eliminate weak-encryption cipher suites from those allowed by the Web server. Attempts to use weak encryption will then fail, with the consequential display of an error page explaining the need to upgrade the browser to 128-bit (strong encryption).

Some service providers may see customer inconvenience issues in requiring a move up to 128-bit browser versions. But the move is really a win-win situation for the customer, because:

- They will always get 128-bit encryption at any site that supports it.
- Newer browsers are more efficient, because they use the latest versions of HTTP and other protocols.

See Also:

- `http://www.verisign.com/` for general information about Verisign
- `http://www.verisign.com/products/site/secure/index.html` for information on Verisign Secure Site Services
- `http://www.verisign.com/products/site/secure/Secure-Site.pdf` for a discussion of differences between 128-bit and 40-bit certificates

Performance Issues

The amount of CPU resources required to accommodate varying loads, both with and without security, is an important best practices issue. When developing applications where cryptography is required, it is important to get an early estimation regarding the CPU and other resources required for volume production systems. Applications are often developed without adding SSL until late in the development cycle, and unpleasant performance surprises frequently occur. Using HTTPS with SSL can increase CPU requirements by 10 to 100 times, compared to HTTP without SSL.

Reverse proxies and special HTTP-to-HTTPS conversion hardware appliances offer the prospect of significant performance gains by shifting SSL processing away from Web servers and Web caches. Oracle is currently investigating these options and will report on them in a later version of *Oracle9i Application Server Best Practices*.

Below are some rules of thumb for SSL in general, relevant for conventional Wintel and Sun processors of around 200 MHz. Because SSL can run on many different platform types in many different implementations, our predicted results should be taken as neither definitive nor particularly accurate for any particular Oracle product. Your results may vary by one half to one order of magnitude.

- Measured in terms of HTTP versus HTTPS pages per second, use of SSL can slow down sites by as much as two orders of magnitude.

  This assumes only one modest-sized (15K or less) HTTP or HTTPS message is sent every few minutes. When many pages are sent or received from a single browser in a small time period (two minutes or less), caching of bulk-encryption keys can reduce the performance difference between HTTP and HTTPS. But you should expect at least a 10-to-1 slowdown when using HTTPS for normal traffic, and you should *always* test reasonable load scenarios.

- Use of SSL adds a latency of several hundred milliseconds, in part because of additional CPU utilization and in part because significant extra network traffic occurs. However, this does not directly translate into throughput numbers.

- Bulk-encryption key size makes little difference in performance when using RC4, because it always uses 128-bit operations. With a 40-bit key, we just know what the other 88 bits in the key are.

- RC4 bulk encryption is about 8 times faster than DES and about 25 times faster than 3DES. Even with a slow Pentium (150MHz), RC4 encrypts 8.5M bytes per second or about 2 milliseconds for a modest-sized (15K) message.

SEC-17: Performance Testing

Performance test applications during development.

Performance testing *with SSL* early in the development cycle is prudent. Test during the prototype or feasibility stages if possible. Testing should emulate volume production.

SEC-18: Sequential HTTPS Transfers

Ensure that sequential HTTPS transfers are requested of the same Web server.

Expect several hundred milliseconds to be required to initiate SSL sessions on a 300 MHz machine. Most of this CPU time is spent in the key exchange logic, where the bulk-encryption key is exchanged. Caching the bulk-encryption key will significantly reduce CPU overhead on subsequent accesses, provided that the accesses are routed to the *same Web server.*

SEC-19: Separate Virtual Servers

Keep secure pages and pages not requiring security on separate virtual servers.

While it may be easier to place all pages for an application on one HTTPS virtual server, the performance cost is very high. Reserve your HTTPS virtual server for pages needing SSL and put the pages not needing SSL on an HTTP virtual server.

If secure pages are composed of many gif, jpeg, or other files that would be displayed on the same screen, then it is probably not worth the effort to segregate secure from non-secure static content. The SSL key exchange (the major consumer of CPU cycles) is likely to be called exactly once in any case, and the overhead of bulk encryption is not that high.

Note: A single Oracle9*i*AS server can accommodate many virtual servers. Some of these can be HTTPS virtual servers, and others can be HTTP virtual servers.

See Also:

`http://isglabs.rainbow.com/isglabs/shawn/SSL_Perf/otpssl8.html` provides a good reference for HTTP versus HTTPS performance

Client Certificates

In the near future, the identity of browser users will be authenticated more and more frequently by client certificates which comply with ITU X.509 version 3 specification. Client certificates contain the following information:

- Expiration date
- Owner's public key
- Owner's distinguished name (including organizational information)
- Trust point (issuer's name) which guarantees that the information in the certificate is valid

Certificates are issued by a certificate authority (CA). Certificates are often issued in-house by large companies, but CA services can also be outsourced. As of January 2001, Verisign is the largest outsourcing agency of certificates, with about 90% of the market. Other players in the market include Thawte, EnTrust, and GTE Cybertrust (part of Baltimore Technologies).

CA standard practice is first to issue certificates and then to keep revocation lists. This allows certificate validation to be handled in the same manner as credit card validation. That is, a certificate is presumed valid if *both* of these are true:

- It is not expired
- It is not on a revocation list

Revocation lists are usually maintained at a central server managed by the CA. They can also be replicated or partially replicated, depending on the security policies of the CA.

Certificates are relatively new instruments for providing user authentication, and firms may be tempted to build policies for them similar to those already in place for usercodes. The traditional usercode model works well for intranets, but it may work poorly in the evolving world of outsourced Internet services. If each service provider were to create its own certificate building rules, then users would be forced to have many different certificates, one for each of the provided services.

It would be very difficult for users to manage multiple certificates within one organization with the infrastructure currently available. The difficulty level would rise even higher if multiple certificates were required for a single transaction (because it included services from a number of organizations). When creating and managing certificates, assume that users will have one certificate for personal use and at most one for each organization where they work.

The Oracle HTTP Server has facilities for certificate handling, including revocation lists. The Oracle HTTP Server allows use of certificates for authentication via SSL and for authorization using the Oracle HTTP Server's URL Wildcard scheme. URL Wildcards are specified with an authorization that can depend upon any of the fields of authenticated certificates. Certificates received via the SSL mechanism encounter a series of checkpoints, each of which must be passed before proceeding to the next:

1. The expiration date is checked.
2. The issuer is checked against the list of valid trust points configured for that Oracle HTTP Server.
3. The revocation list is checked to ensure that the certificate is not on it.
4. The URL is examined for matches against various configured URL Wildcards.

   Each URL Wildcard has a list of allowable valid certificate patterns. A certificate pattern can include any of the fields of a certificate. This could include the issuer, the organizational unit of the owner, the owner's name, or fragments of any of these fields.
5. If a match occurs, then the operation proceeds, and the URL is processed.

Authentication using X.509 certificates (rather than usercode/password) offers several interesting possibilities, especially when viewed from the perspective of the relatively new market of outsourced CA services:

- Certificates could be handled like traditional usercodes.

  Each user (whose identity would include organizational unit and issuer fields) would be explicitly entered in access control lists. Membership in such lists would be required to allow access to protected services.
- Particular certificate fields (such as the issuer or organizational unit) could be used to authorize access to services without regard to the certificate holder's distinguished name.

  Thus, Gartner might allow anyone at Oracle to access their market research information, even though they might not specifically know that the person with a particular distinguished name was an Oracle employee. Also, Oracle might allow anyone whose issuer was "Oracle" to have access to certain Oracle services.

- A revocation list can be used several ways.

  It could be used, for example, only to exclude certificates whose private keys were compromised or for people who left an organization. On the other hand, it might be used instead to remove both invalid users (compromised keys and those who leave) and otherwise valid users (employees on probation). There is nothing which prevents organizations from adding and deleting certificates from revocation lists.

SEC-20: Organization Identity

Ensure that certificate organization unit plus issuer fields uniquely identify the organization across the Internet.

One way to accomplish this would be to include the Dun and Bradstreet or IRS identification as identification for the issuer and the organizational unit within the certificate.

SEC-21: User Identity

Ensure that certificate issuer plus distinguished name uniquely identify the user.

If the combination of issuer and distinguished name is used as identification, then there must be no chance of duplication. Note that authentication based on the public key may be a poor idea, because you would have to revoke it if the private key were compromised. Public key authentication is risky even if you expect to use the certificate only within an intranet, because you may later decide to outsource services.

SEC-22: Expiring Certificates

Include expiring certificates in tests of applications using certificates.

Expiration is an important consideration for a number of reasons. Unlike most usercode-based systems, certificates expire automatically. The expiration period is typically (but not necessarily) one year. With longer-duration certificates, fewer reissues are required, but revocation lists become larger.

Expiring certificates can become time bombs of bugs, if they are not included in tests of application systems using certificates. Consider the following examples:

- If processing is distributed over several time zones, then certificates might be simultaneously authorized on one system and not on others, due to differences in system clock and assumed GMT.

- The replay of a transaction might fail if a certificate expires between the initial transaction and the replay.
- An expiring certificate might provide authorization during early stages of a transaction but fail later.

In systems where certificates replace traditional usercodes, these situations may result in unexpected bugs. Careful consideration of the effects of expiration is required. You will need to develop new policies, because most application and infrastructure developers have not yet worked in systems where authorization might change during transactions.

SEC-23: Certificate Reissues

Use certificate reissues to update certificate information.

Because certificates expire, infrastructure for updating expired certificates will be required. Take advantage of the reissue to update organizational unit or other fields.

In cases of mergers, acquisitions, or status changes of individual certificate holders, consider reissue even when the certificate has not yet expired. But pay attention to key management. If the certificate for a particular person is updated before it expires, for example, then should the old certificate automatically be put on the revocation list?

SEC-24: Certificate Revocations

Audit certificate revocations.

Revocation audit trails can help you reconstruct the past when necessary. An important example is replay of a transaction to ensure the same results on the replay as during the original processing. If the certificate of a transaction participant was revoked between the original and the replay, then the audit trail enables certificate "unrevocation".

Note: The quote marks around *unrevocation* carry a message. We are not recommending actual unrevocation operations, which can significantly complicate the management of systems employing certificates. It may be best to disallow unrevocation operations in production systems entirely, even though certificate infrastructure may allow them.

EXPORT versus DOMESTIC Encryption Issues

For years U.S. developers have had to cope with building DOMESTIC and EXPORT versions of products containing cryptographic technology. In many cases, getting products through the complex process of certification for export was so difficult that it limited other areas of functionality as well.

Export Law Changes

Changes to U.S. export law in early 2000 eliminated the requirement for EXPORT and DOMESTIC versions of products containing cryptographic functions. But each new version of software containing cryptographic functions must still have a one-time U.S. government review before shipment outside the U.S. and Canada.

> **Note:** While general guidelines are provided here, all products containing any cryptographic logic should be reviewed by corporate export control.

As of June 2001, rules for products containing encryption are roughly:

- Products that are never to be distributed outside the U.S. and Canada can be released without government review.
- No shipments of products containing cryptographic functions are allowed to countries deemed "sponsors of international terrorism" by the U.S. State Department, without specific State Department approval. As of May 2001, these countries include: Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria.
- New versions of products planned for delivery to other (that is, non-terrorist) countries outside the U.S. and Canada may require a one-time review by U.S. State and Commerce Departments (or their agents), depending on the cryptographic functions which they employ.
- Products containing no encryption or whose encryption is limited to nondata items like usercodes and passwords can be shipped without restrictions, licenses, or reviews.

  Examples of cryptographic functions that can be used without restriction or license include digital signatures and authentication technology in general. These functions are exempt because they could not easily be adapted for keeping the *content* of messages or data confidential.

- Products containing encryption technology that could be used to send encrypted data or messages must be reviewed. This includes new versions of such code any time new or substantially changed versions of cryptography functions are issued.

These reviews generally take six weeks or less if new cryptographic techniques are not employed. Using encryption technology such as SSL (which has already been reviewed) will usually shorten the review. Developing or using a new encryption technology or a new usage of existing technology can lengthen the reviews to many months.

Note: Shipments to military organizations and some other governmental organizations may require special licenses for every shipment. This should not be an issue for developers as long as the one-time review has been performed.

See Also:

`http://cwis.kub.nl/~frw/people/koops/lawsurvy.htm` for a very good reference on international cryptography laws

5

Using Oracle9*i*AS Web Cache with Third-Party Servers

This chapter first provides an overview of Oracle9*i*AS Web Cache and discusses its integration with third-party application servers from a general perspective, followed by detailed examples involving three particular application servers.

This chapter contains these topics:

- Oracle9iAS Web Cache Overview
- Integration with Third-Party Application Servers
- BEA WebLogic 6.0 Java Applications
- IBM WebSphere 3.5.2 Java Applications
- Microsoft IIS 5.0 ASP Applications

> **Note:** The application examples used in the discussions of these third-party servers are relatively simple. Running with production applications will usually require more extensive configuration of Oracle9*i*AS Web Cache.

Oracle9*i*AS Web Cache Overview

Oracle9*i*AS Web Cache is the industry's first caching solution designed from the ground up for e-business. With its unique combination of server acceleration and server load balancing, Oracle9*i*AS Web Cache tears down the performance barriers that plague today's dynamic e-business Web sites. Unlike legacy cache servers which handle only static data, Oracle9*i*AS Web Cache accelerates the delivery of both static and dynamically generated Web content, thereby improving response times for feature-rich pages.

As the first and only caching solution on the market to support Edge Side Includes (ESI) for performing page assembly in edge servers, Oracle9*i*AS Web Cache leads the industry with its ability to deliver rich, personalized content from both the edge of the data center and the edge of the Internet.

Deployed before a farm of application Web servers or globally at the network edge, Oracle9*i*AS Web Cache also provides load balancing, failover, and patent-pending surge protection features for Web servers. These features combine to ensure blazing Web-site performance and rock-solid uptime while reducing the cost of doing business online. With Oracle9*i*AS Web Cache, e-businesses can now serve compelling content faster, to more customers, using fewer computing resources than ever before.

Integration with Third-Party Application Servers

Oracle9*i*AS Web Cache sits in front of application Web servers, caching their content and providing that content to Web browsers that request it. Oracle9*i*AS Web Cache intercepts all requests from the Web browser to the Web site. If the requested objects are present in the cache, then Oracle9*i*AS Web Cache sends them to the Web browser. If the requested objects are not in the cache (known as a cache miss), then Oracle9*i*AS Web Cache forwards the requests via HTTP to the Web server.

Because Oracle9*i*AS Web Cache is transparent to the Web server, the Web server treats HTTP requests from Oracle9*i*AS Web Cache as any other HTTP request coming directly from the browser client. In turn, the Web server generates the response (for example using ASP, PSP, or Java servlet) and sends it back to Oracle9*i*AS Web Cache as an HTTP message.

Oracle9*i*AS Web Cache then caches any cacheable objects returned by the Web server and forwards the response back to the browser. As Oracle9*i*AS Web Cache becomes populated, it is able to serve more of the requested content itself, freeing up processing resources on application Web servers and database servers.

Because Oracle9*i*AS Web Cache fully supports HTTP, it can work with any HTTP-compliant application Web server. How the application Web servers choose to generate HTTP responses is irrelevant to Oracle9*i*AS Web Cache.

The type of Web server that a site uses depends mainly on the types of applications that site is running. For example, if customers want to run Active Server Pages (ASP), then they may prefer to use Microsoft Internet Information Server (IIS) as the Web server.

This section contains these topics:

- Web-site Configuration
- Cacheability Rules and Expiration Rules

Web-site Configuration

You configure Oracle9*i*AS Web Cache to communicate with a third-party application Web server the same way you would with Oracle HTTP Server, by providing the host name and the listening port number. The default values for the listening ports for the products discussed in this chapter are given in Table 5–1.

Table 5–1 Application Server Default Listening Ports

Application Web Server	Port
BEA WebLogic 6.0	7001
IBM HTTP Server / IBM WebSphere 3.5.2	80
Microsoft IIS 5.0	80

To configure Oracle9*i*AS Web Cache to communicate with a third-party application Web server:

1. Start Oracle9*i*AS Web Cache. From the command line, enter:

   ```
   webcachectl start
   ```

2. Start Oracle Web Cache Manager. Point your browser to:

   ```
   http://web_cache_hostname:4000/webcacheadmin
   ```

   When prompted, enter `administrator` for the administrator user name and `administrator` for the password (the first time you log in).

3. In the navigator pane, select Administering Web Sites > Application Web Servers.

   The Application Web Servers page appears in the right pane.

4. In the Application Web Servers page, click Add.

   The Edit / Create Application Web Server page dialog box appears.

5. In the Hostname field, enter the hostname or IP of the application Web server.

6. In the Port field, enter the listening port from which the application Web server will receive Oracle9*i*AS Web Cache requests.

7. In the Capacity field, enter the number of concurrent connections that the application Web server can sustain.

8. Leave Failover Threshold as is.

9. In the Ping URL field, enter the URL that Oracle9*i*AS Web Cache will use to poll the application Web server that has reached its failover threshold.

10. Leave the Ping Interval as is. The default value is 10 seconds.

11. Click Submit.

12. In the Oracle Web Cache Manager main window, click Apply Changes.

13. In the navigator pane, select Administering Oracle Web Cache > Web Cache Operations.

    The Oracle Web Cache Operations page appears in the right pane.

14. In the Oracle Web Cache Operations page, click Stop and then Start to restart Oracle Web Cache.

Note: Oracle9*i*AS Web Cache may be deployed on the same computer as your Web servers or on a dedicated computer in front of your Web server farm. For more information on configuring or modifying security settings or configuring additional listening ports from which Oracle9*i*AS Web Cache will receive browser requests, refer to the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

Cacheability Rules and Expiration Rules

You assign cacheability rules and expiration rules when using third-party application Web servers in the same way as when using Oracle HTTP Server. You can select to cache or not to cache content for:

- Static documents
- Multiple-version URLs
- Personalized pages
- Pages that support session tracking
- HTTP error messages

You can also assign an expiration time limit to documents or invalidate documents at any time.

The following sections describe how to specify cacheability rules and expiration rules for each example. For more information about specifying rules, refer to the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

BEA WebLogic 6.0 Java Applications

For this demonstration, we used an evaluation copy of WebLogic 6.0 running on Microsoft Windows NT and Oracle9*i*AS Web Cache release 2.0.0. The WebLogic 6.0 installation includes a number of JSP, servlet, and EJB examples. For the purposes of this demonstration, we used:

- WebLogic SnoopServlet
- WebLogic SessionServlet

WebLogic SnoopServlet

SnoopServlet demonstrates getting and using request information, headers, and parameters sent by the browser. We will use it to demonstrate how Oracle9*i*AS Web Cache caches full-page dynamic content.

Before you start, ensure that Oracle9*i*AS Web Cache has been configured to communicate with the WebLogic 6.0 Application Server as described in "Web-site Configuration" on page 5-3. Next, start the WebLogic 6.0 Examples Server and access the following URL:

```
http://hostname:7001/examplesWebApp/SnoopServlet
```

When you access the URL, notice that your browser displays request information, headers, parameters, and the GIF image "Build On bea".

To cache this content, you need two rules: one to cache the GIF image and one to cache the SnoopServlet output. The rule for the GIF is configured as a default rule with Oracle9*i*AS Web Cache release 2.0.0. To create the rule for the SnoopServlet output, follow procedure "Configuring Cacheability Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

When creating the cacheability rule, ensure that the following steps are performed:

1. In the Cacheability Rules page, insert the rule at the top of the rule list.

   The Edit Cacheability Rule dialog box appears.

2. In the URL Expression field, enter: `/examplesWebApp/SnoopServlet`
3. In the HTTP Method(s) row, choose GET.
4. Select Cache.
5. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

As a part of this demonstration, verbose mode will be needed in the Web Cache Event Log settings. To turn on verbose event logging:

1. In the navigation pane of the Oracle Web Cache Manager, select Administering Oracle Web Cache > Event Logging.

   The Event Logging page appears.

2. Choose Edit in the Event Logging page.

   The Change Options for Event Logging dialog box appears.

3. In Verbose Logging, select Yes.
4. Apply changes and restart Oracle9*i*AS Web Cache from the Oracle Web Cache Operations page, as described in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

Now, rather than pointing your browser to the WebLogic 6.0 Application Server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/examplesWebApp/SnoopServlet
```

The output is the same as it was when you accessed the SnoopServlet directly from the WebLogic 6.0 Application Server. But this time, Oracle9*i*AS Web Cache caches the SnoopServlet output and serves the request to the client.

One way to verify this is to look at the Oracle9*i*AS Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made in the log based on the cacheability rules defined in the previous steps:

```
06/Jun/2001:08:56:13 -0800 -- uri /examplesWebApp/SnoopServlet matches
cacheable rule /examplesWebApp/SnoopServlet
06/Jun/2001:08:56:13 -0800 -- cache inserted one file
'/examplesWebApp/SnoopServlet gzip' in bucket 31450
06/Jun/2001:08:56:13 -0800 -- uri /examplesWebApp/images/BEA_Button_Final_
web.gif matches cacheable rule \.gif$
06/Jun/2001:08:56:13 -0800 -- cache inserted one file
'/examplesWebApp/images/BEA_Button_Final_web.gif' in bucket 86754
```

From this point on, anytime a client accesses the SnoopServlet, the response will be served from Oracle9*i*AS Web Cache.

WebLogic SessionServlet

SessionServlet provides a simple example of an HTTP servlet that uses the HttpSession class to track the number of times that a client has visited the servlet. We will use it to demonstrate how Oracle9*i*AS Web Cache caches pages with session-encoded URLs.

Before you run this example, ensure that Oracle9*i*AS Web Cache has been configured to communicate with the WebLogic 6.0 Examples Server as described in "Web-site Configuration" on page 5-3.

Next, configure your browser not to accept cookies. This is required in order to use session-encoded URLs in this example. Finally, start the WebLogic 6.0 Examples Server and access the following URL:

```
http://hostname:7001/examplesWebApp/SessionServlet
```

When you access the URL, notice that the page displays how many times a client has visited it. When you click the link labeled "here", notice that the session ID is encoded in the URL. Every time you refresh or reload the page, the counter increases by one.

To cache the content of the "here" page, create expiration and session-related caching rules for the SessionServlet output.

To create the expiration rule, follow procedure "Configuring Expiration Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. In the Create Expiration Rule dialog box, elect to expire the output 60 seconds after cache entry. In the After Expiration section, select Remove immediately.

To create a session-related caching rule, follow procedure "Configuring Rules for Pages with Session Tracking" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. When configuring a session-related caching rule, ensure that the following steps are performed:

- Create a session definition in the Edit > Create Session > Personalized Attribute Definition dialog box:
  1. In the Session / Attribute field, enter `BEASession`.
  2. In the Cookie Name field, enter `JSESSIONID`.

     `JSESSIONID` is the default cookie name used by the WebLogic 6.0 Application Server.
  3. In the URL Parameter field, enter `jsessionid`.
- In the Add Session Related Caching Rule dialog box:
  1. From the Please select a session / attribute list, select BEASession.
  2. Select YES for prompt 1.
  3. Select YES for prompt 2.
  4. Select NO for prompt 3.
- Create a new cacheability rule for `/examplesWebApp/SessionServlet`.
  1. In the Cacheability Rules page, insert a new rule at the top of the rule list.

     The Edit Cacheability Rule dialog box appears.
  2. In the URL Expression field, enter: `/examplesWebApp/SessionServlet`
  3. In the HTTP Method(s) row, choose GET.
  4. Select Cache.
  5. In the Expiration Rule row, select "Expire: 60 seconds in cache" and "After: remove immediately"

6. In the Session / Personalized Attribute Related Caching Rules row, select "Apply the following" and "BEASession: cache with session, cache w / o session"
7. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

Turn on verbose event logging, apply changes, and restart Oracle9*i*AS Web Cache, as described in "WebLogic SnoopServlet" on page 5-5.

Now, rather than pointing your browser to the WebLogic 6.0 Application Server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/examplesWebApp/SessionServlet
```

The output is the same as it was when you accessed the SessionServlet servlet directly from the WebLogic 6.0 Application Server. This time Oracle9*i*AS Web Cache caches the SessionServlet output. When the page is refreshed or reloaded, notice that the counter does not increment by one. This is because Oracle9*i*AS Web Cache serves the content, and the request never goes to the WebLogic 6.0 Application Server.

Another way to verify this is to look at the Oracle9*i*AS Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made in the log based on the cacheability rules defined in the previous steps:

```
06/Jun/2001:14:29:45 -0800 -- uri
/examplesWebApp/SessionServlet;jsessionid=OtS9hLGk0sB8mZdM9NpIs2kFQuHIkhPmjz
i3i46zt2HCY22gXf65!1555932292067982192!-1979543882!7001!7002 matches
cacheable rule /examplesWebApp/SessionServlet
06/Jun/2001:14:29:45 -0800 -- cache inserted one file
'/examplesWebApp/SessionServlet;jsessionid=@ JSESSIONID'in bucket 51355
```

When reloading the page, you should also notice that the cached response appears faster than when you access the WebLogic server directly.

Because the expiration rule for this URL is set to 60 seconds, Oracle9*i*AS Web Cache will expire the cached content after 60 seconds and refresh the content the next time the user requests the page.

IBM WebSphere 3.5.2 Java Applications

For this demonstration, we used an evaluation copy of WebSphere 3.5.2 running on Microsoft Windows NT and Oracle9*i*AS Web Cache release 2.0.0. The WebSphere installation includes a number of JSP, servlet, and EJB examples. For the purposes of this demonstration, we used:

- WebSphere Snoop Servlet
- WebSphere SessionSample

WebSphere Snoop Servlet

WebSphere Snoop Servlet demonstrates getting and using request information, headers, and parameters sent by the browser. With this example, we will demonstrate how Oracle9*i*AS Web Cache caches full-page dynamic content.

Before you start, ensure that Oracle9*i*AS Web Cache has been configured to communicate with the WebSphere 3.5.2 Application Server as described in "Web-site Configuration" on page 5-3.

Next, start the WebSphere 3.5.2 server and access the following URL:

```
http://hostname/servlet/snoop
```

When you access the URL, notice that request information, headers, and parameters sent by your browser are displayed.

To cache this content, create a cacheability rule for the WebSphere Snoop Servlet output. To create the rule, follow the procedures for "Configuring Cacheability Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

When creating the cacheability rule, ensure that the following steps are performed:

1. In the Cacheability Rules page, insert a new rule at the top of the rule list.

   The Edit Cacheability Rule dialog box appears.
2. In the URL Expression field, enter: `/servlet/snoop`
3. In the HTTP Method(s) row, choose GET.
4. Select Cache.
5. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

Turn on verbose event logging, accept changes, and restart Oracle9*i*AS Web Cache, as described earlier in "WebLogic SnoopServlet" on page 5-5.

Now, rather than pointing your browser to the WebSphere 3.5.2 Application Server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/servlet/snoop
```

The output is the same as it was when you accessed the WebSphere Snoop Servlet directly from the WebSphere Application Server. This time, Oracle9*i*AS Web Cache caches the output and serves the response to your browser. One way to verify this is to look at the Oracle9*i*AS Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made into the log file based on the cacheability rule defined in the previous steps:

```
13/Jun/2001:08:33:23 -0800 -- uri /servlet/snoop matches cacheable rule
/servlet/snoop
13/Jun/2001:08:33:23 -0800 -- cache inserted one file '/servlet/snoop gzip' in
bucket 40796
```

When you reload the page, you should notice that the cached response appears faster than when you access the WebSphere server directly.

WebSphere SessionSample

SessionSample is a simple example of an HTTP servlet that tracks the number of times that a client has visited the servlet using a cookie. With this example, we will demonstrate how Oracle9*i*AS Web Cache caches pages with session cookies.

This example is not a pre-deployed WebSphere example like Snoop Servlet. You can find this example in "section 4.4.1.1: Session programming model and environment" of the WebSphere 3.5 online documentation, when you click on the `SessionSample.java` link on that page.

To run this example, first compile the `SessionSample.java` file in your WebSphere environment and then copy the `SessionSample.class` file in the location where the `snoop.class` file resides. The default location for the `snoop.class` file is WebSphere's install directory:

```
\WebSphere\AppServer\hosts\default_host\default_app\serv lets\
```

Before you run this example, start WebSphere 3.5.2, set your browser to accept cookies, and access the following URL:

```
http://hostname/servlet/SessionSample
```

When you access the URL, notice that the page displays the number of times a client has visited this page. When you reload this page, the counter increments by one.

To cache the content of this page, you create expiration and session-related caching rules for the SessionSample servlet output. To do that, you need to know the cookie name that the SessionSample servlet is using. The default cookie name used by the WebSphere 3.5.2 Application Server is `sesessionid`. The same cookie name is used in this example.

To create the expiration rule, follow procedure "Configuring Expiration Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. In the Create Expiration Rule dialog box, elect to expire the output 60 seconds after cache entry. In the After Expiration section, select Remove immediately.

To create a session-related caching rule, follow procedure "Configuring Rules for Pages with Session Tracking" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. When configuring a session-related caching rule, ensure that the following steps are performed:

- Create a session definition in the Edit > Create Session > Personalized Attribute Definition dialog box:
  1. In the Session / Attribute field, enter `BEASession`.
  2. In the Cookie Name field, enter `JSESSIONID`.

     `JSESSIONID` is the default cookie name used by the WebLogic 6.0 Application Server.
  3. In the URL Parameter field, enter `jsessionid`.
- In the Add Session Related Caching Rule dialog box, perform the following:
  1. From the Please select a session / attribute list, select BEASession.
  2. Select YES for prompt 1.
  3. Select YES for prompt 2.
  4. Select NO for prompt 3.

- Create a new cacheability rule for `/examplesWebApp/SessionServlet`:
  1. In the Cacheability Rules page, insert a new rule at the top of the rule list.

     The Edit Cacheability Rule dialog box appears.
  2. In the URL Expression field, enter `/examplesWebApp/SessionServlet`.
  3. In the HTTP Method(s) row, choose GET.
  4. Select Cache.
  5. In the Expiration Rule row, select: "Expire: 60 seconds in cache" and "After: remove immediately"
  6. In the Session / Personalized Attribute Related Caching Rules row, select: "Apply the following" and "BEASession: cache with session, cache w/o session".
  7. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

Turn on verbose event logging, accept changes, and restart Oracle9*i*AS Web Cache, as described in "WebLogic SnoopServlet" on page 5-5.

Now, rather than pointing your browser to the WebSphere 3.5.2 Application Server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/servlet/SessionSample
```

The output is the same as when you access the SessionSample servlet directly from WebSphere 3.5.2. This time, Oracle9*i*AS Web Cache caches the SessionSample servlet output. To verify that the content is served by the cache, refresh or reload the page. Notice that the counter remains the same. This is because Oracle9*i*AS Web Cache serves the content, and the request never goes to WebSphere 3.5.2.

Another way to verify this is to look at the Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made into the log based on thecacheability rules defined in the previous steps:

```
13/Jun/2001:08:25:54 -0800 -- uri /servlet/SessionSample matches cacheable rule
/servlet/SessionSample
13/Jun/2001:08:25:54 -0800 -- cache inserted one file '/servlet/SessionSample
sesessionid;gzip' in bucket 72582
```

When you reload the page, you should also notice that the cached response appears faster than when you access the WebSphere server directly.

The content will expire 60 seconds from the time it was cached, and Oracle9*i*AS Web Cache will update it the next time it is requested.

Microsoft IIS 5.0 ASP Applications

For this demonstration, we used an evaluation copy of Microsoft IIS 5.0 running on Microsoft Windows 2000 Server and Oracle9*i*AS Web Cache release 2.0.0. The installation includes a number of ASP examples. For the purpose of this chapter, we used:

- ServerVariables_Jscript ASP
- Cookie_Jscript ASP

ServerVariables_Jscript ASP

ServerVariables_JScript.asp demonstrates techniques you can use to access server variable information from an ASP script. With this example, we will demonstrate how Oracle9*i*AS Web Cache caches full-page dynamic content.

Before you start, ensure that Oracle9*i*AS Web Cache has been configured to communicate with the IIS 5.0 server as described "Web-site Configuration" on page 5-3.

Next, start the IIS 5.0 server and access the following URL:

```
http://hostname/IISSamples/sdk/asp/interaction/ServerVar iables_JScript.asp
```

When you access the URL, notice that request information, headers, and parameters sent by your browser are displayed.

To cache this content, create a cacheability rule for the ServerVariables_Jscript ASP output. To create the rule, follow procedure "Configuring Cacheability Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library.

When creating the cacheability rule, ensure that the following steps are performed:

1. In the Cacheability Rules page, insert a new rule at the top of the rule list.

   The Edit Cacheability Rule dialog box appears.

2. In the URL Expression field, enter:

   ```
   /IISSamples/sdk/asp/interaction/ServerVariables_JScript.asp
   ```

3. In the HTTP Method(s) row, choose GET.

4. Select Cache.
5. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

Turn on verbose event logging, accept changes, and restart Oracle9*i*AS Web Cache, as described in "WebLogic SnoopServlet" on page 5-5.

Now, rather than pointing your browser to the Microsoft IIS 5.0 Application Server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/IISSamples/sdk/asp/interaction/ServerVariables_
JScript.asp
```

The output is the same as it was when you accessed the ServerVariables_Jscript ASP directly from the IIS server. This time, Oracle9*i*AS Web Cache caches the ServerVariables_Jscript ASP output and serves the request to the client.

To verify this, look at the Oracle9*i*AS Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made in the log file based on the cacheability rule defined in the previous steps:

```
06/Jun/2001:05:07:39 -0800 -- uri
/IISSamples/sdk/asp/interaction/ServerVariables_JScript.asp matches
cacheable rule /IISSamples/sdk/asp/interaction/ServerVariables_JScript.asp
06/Jun/2001:05:07:39 -0800 -- cache inserted one file
'/IISSamples/sdk/asp/interaction/ServerVariables_JScript.asp gzip' in bucket
12173
```

Cookie_Jscript ASP

Cookie_JScript.asp illustrates how your script can set and read cookies by using the Response.Cookies collection. With this example, we will demonstrate how Oracle9*i*AS Web Cache caches pages with session cookies.

Before you run this example, ensure that Oracle9*i*AS Web Cache has been configured to communicate with the IIS 5.0 server as described in "Web-site Configuration" on page 5-3.

Next, start the IIS 5.0 server, verify that your browser is set to accept cookies, and access the following URL:

```
http://hostname/IISSamples/sdk/asp/interaction/Cookie_JS cript.asp
```

When you access the URL, notice that the page displays the date and time you last visited this page. When you click "Revisit this page", the date and time is updated.

To cache the content of this page, create expiration and session-related caching rules for the Cookie_Jscript output. To do that, you need to know the cookie name that the Cookie_Jscript ASP is using. The default cookie name used by the Microsoft IIS ASP applications is `ASPSESSIONID`. The cookie name used in this example is `CookieJScript`.

To create the expiration rule, follow procedure "Configuring Expiration Rules" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. In the Create Expiration Rule dialog box, elect to expire the output 60 seconds after cache entry. In the After Expiration section, select Remove immediately.

To create a session-related caching rule, follow procedure "Configuring Rules for Pages with Session Tracking" in "Creating Rules for Cached Content" in the *Oracle9iAS Web Cache Administration and Deployment Guide* in the Oracle9*i* Application Server Documentation Library. When configuring a session-related caching rule, ensure that the following steps are performed:

- ❏ Create a session definition in the Edit > Create Session > Personalized Attribute Definition dialog box:
  1. In the Session / Attribute field, enter `MSSession`.
  2. In the Cookie Name field, enter `CookieJSCript`.
- ❏ In the Add Session Related Caching Rule dialog box, perform the following:
  1. From the Please select a session / attribute list, select MSSession.
  2. Select YES for prompt 1.
  3. Select YES for prompt 2.
  4. Select NO for prompt 3.
- ❏ Create a new cacheability rule for the URL `/IISSamples/sdk/asp/interaction/Cookie_JScript.asp`
  1. In the Cacheability Rules page, insert a new rule at the top of the rule list.

     The Edit Caceability Rule dialog box appears.
  2. In the URL Expression field, enter:

     `/IISSamples/sdk/asp/interaction/Cookie_JScript.asp`

3. In the HTTP Method(s) row, choose GET.
4. Select Cache.
5. In the Expiration Rule row, select: "Expire: 60 seconds in cache" and "After: remove immediately"
6. In the Session / Personalized Attribute Related Caching Rules row, select: "Apply the following" and "MSSession: cache with session, cache w / o session"
7. Leave all other defaults in the Edit Cacheability Rule dialog box as is.

Turn on verbose event logging, accept changes, and restart Oracle9*i*AS Web Cache, as described in "WebLogic SnoopServlet" on page 5-5.

Now, rather than pointing your browser to the IIS 5.0 server, point your browser to Oracle9*i*AS Web Cache and access the following URL:

```
http://web_cache_hostname:1100/IISSamples/sdk/asp/interaction/Cookie_JScript.asp
```

The output is the same as it was when you accessed the Cookie_JScript directly from Microsoft IIS. This time, Oracle9*i*AS Web Cache caches the Cookie_JScript output. To verify that the cache serves the content, click "Revisit this page". Notice that the date and time are not updated. This is because Oracle9*i*AS Web Cache serves the cached content, and the request never goes to Microsoft IIS.

Another way to verify this is to look at the Oracle9*i*AS Web Cache Event Log file:

```
$ORACLE_HOME/webcache/logs/event_log
```

In our test, the following entries were made into the log based on the cacheability rules defined in the previous steps:

```
06/Jun/2001:04:32:34 -0800 -- uri /IISSamples/sdk/asp/interaction/Cookie_
JScript.asp matches cacheable rule /IISSamples/sdk/asp/interaction/Cookie_
JScript.asp
06/Jun/2001:04:32:34 -0800 -- cache inserted one file
'/IISSamples/sdk/asp/interaction/Cookie_JScript.aspgzip' in bucket 92163
```

The content will expire 60 seconds from the time it was cached, and Oracle9*i*AS Web Cache will update it the next time it is requested.

Glossary

bastion hosts

Well-fortified servers running initial point of contact protocols, such as HTTP or SMTP.

hash

Hashing is a scheme for providing rapid access to data items which are distinguished by some key. Each data item to be stored is associated with a key. A hash function is applied to the item's key and the resulting hash value is used as an index to select one of a number of "hash buckets" in a hash table. The table contains pointers to the original items.

idempotent

A function is idempotent if repeated applications have the same effect as one.

load balancers

Balancing, failure detecting, multi-layer (OSI network reference model layers 2-7) switches which can detect failure of clustered mid-tier servers and route around the failed servers. Load balancers also allow mid-tier servers to be taken out of service gracefully. Once out of service, a mid-tier server may be reconfigured or simply rebooted in order to reduce the chance of a crash or slowdown due to resource leaks and other bugs. Example are Cisco's CSS 11000 or F5's BigIP.

pool

A collection of instances of an expensive resource, such as a database connection-session pair, that can be used sequentially by different clients.

serial reuse

Several clients taking turns at using a resource. At any time only one client is using the resource, and only when the client is done with the resource will the resource be used by another client.

stateful inspection

A security system in which a firewall keeps track of various sessions by protocol and ensures that illegal protocol transitions are disallowed through the firewall. This blocks the types of intrusion which exploit illegal protocol transitions.

stateless

Absence of per-client state in a resource. For example, a database connection or session is stateless when all the client database updates have been committed or rolled back.

throttling

Controlling the maximum usage of a shared resource. Throttling is important when the resource is shared by many different applications, and you want to be able to guarantee decent response times for some class of applications or clients.

Index

A

access logging, 2-31
Active Server Pages, 5-3
AllowOverride, 2-32
applications
  and connection caches, 3-5
  and multiple JVMs, 3-13
  BEA WebLogic 6.0 Java, 5-5
  bugs, 1-5
  cryptographic, 4-14
  database-centric, 2-12
  duplicate requests, 1-23
  IBM WebSphere 3.5.2 Java, 5-10
  memory leaks, 2-12
  Microsoft IIS 5.0 ASP, 5-14
  performance optimization, 1-20
  performance testing, 4-15
  resource drains, 1-18
  stateless, 2-5, 2-6
  statesafe, 1-17
  users' view of availability, 1-6
  using certificates, 4-18
Array, 2-25
ArrayList, 2-25
availability
  data, 1-6
  defined, 1-1
  example, 1-4
  hardware, 1-4
  key practices, 1-2
  measuring, 1-3
  software, 1-5, 1-17
  users' view, 1-6

B

backups
  Oracle Internet File System, 1-25
  schedule, 1-24
  standby database, 1-25
bastion hosts
  defined, 4-4
browser upgrade, 4-13
browsers
  and Oracle9iAS Web Cache, 5-2
  cookies, 1-17
  cross-site scripting, 4-9
  DOMESTIC vs. EXPORT, 4-11
  redirect, 2-17
  reload function, 1-7
  SSL support, 4-12
  user identification, 4-16
buffer overflow, 4-8

C

cache
  expiration, 5-5
  mid-tier, 2-11
  miss, 5-2
  rules, 5-5
case study
  description, 2-3
  endRequest, 2-5
  memory leaks, 2-12
  memory overhead, 2-8
  results, 2-7
  session scope bean, 2-7

sessions, 2-7
use of Hashtables, 2-5
user information, 2-4
catch-all exceptions, 1-21
certificates
128-bit, 4-13
40-bit, 4-11
and Oracle HTTP Server, 4-17
audit revocations, 4-19
authority, 4-16
client, 4-16
expiring, 4-18
multiple, 4-16
organization identity, 4-18
reissuing, 4-19
revocation, 4-16
user identity, 4-18
validation, 4-16
X.509, 4-17
classes
final, 2-2
Hashtable, 2-25
String, 2-24
StringBuffer, 2-24
Vector, 2-25
clear(), 2-27
commands
"r", 4-10
commercial drivers, 2-34
configuration
JServ, 2-19
Web site, 5-3
connections
pooling, 2-9
Cookie_JScript.asp, 5-15
cookies, 5-7
cross-site scripting, 4-9

D

data availability, 1-6
database
connections, 2-11
tuning, 2-12
Database Cache, 2-11
definitions
availability, 1-1
load balancers, 1-8
pooling, 2-9
statesafe, 1-17
detecting
failures, 1-1
directives
AllowOverride, 2-32
FollowSymLinks, 2-32
SSLSessionCacheTimeout, 2-31
DMZ
defined, 4-3
services, 4-5
DNS lookup, 2-31
documenting
procedures, 1-28
drivers
commercial, 2-34
dynamic content, 5-14
dynamic include
description, 2-16
Dynamic Monitoring Service, 2-9
and Java heap, 2-20
description, 2-34
sample metrics, 2-34

E

Edge Side Includes, 5-2
encryption
weak, 4-11
endRequest, 2-5
EnTrust, 4-16
equals(), 2-26
error page, 4-13
escape characters, 4-9
evaluating
performance, 2-33
event logging, 5-7
exceptions
catch-all, 1-21
expiring
certificates, 4-18
export

cryptographic software, 4-9
law changes, 4-20
laws, 4-11
reviews, 4-21

F

failures
detecting, 1-1
recording, 1-6
fault containment, 4-8
final class, 2-2
finally clause, 1-22
firewalls
and stateful inspection, 4-5
architecture illustrated, 4-3, 4-7
implementation, 1-9
switched connection, 4-4
tradeoffs, 4-2
FollowSymLinks, 2-32
forward, 2-17
FTP, 4-5

G

Global Server ID Certificate, 4-12
goals
performance, 2-33
GTE Cypertrust, 4-16
guidelines
sessions, 2-8
SSL, 4-14

H

hardware availability, 1-4
hashCode(), 2-26
HashMap, 2-25
Hashtables, 2-25
case study, 2-5
HTTP, 4-5
HTTPS, 4-7

I

identities
organization, 4-18
user, 4-18
IMAP4, 4-5
immutable object, 2-24
implementation
SingleThreadModel, 2-13
information of record, 4-6
initialization
lazy, 2-27

J

Java
bugs, 1-5
code separation, 2-15
performance, 2-21
synchronization, 1-17, 2-21
Java heap
and Dynamic Monitoring Service, 2-20
default, 2-20
setting size, 2-19
JavaServer Pages
debug, 2-21
deployment, 2-18
developer_mode, 2-21
pre-translate, 2-20
JDBC
connections pooling, 2-9
statement caching, 2-10
JServ
autoreload.classes, 2-20
configuration, 2-19
default process, 2-19
multiple, 2-19
performance, 2-19

K

KeepAlive
tuning, 2-30
key practices
availability, 1-2

L

lazy initialization, 2-27
load balancers
  commercial, 1-13
  definition, 1-8

M

MaxClients
  tuning, 2-30
measuring
  availability, 1-3
memory leaks, 2-12
  and Java, 2-12
  case study, 2-12
memory overhead
  case study, 2-8
methodology
  performance, 2-33
methods
  clear(), 2-27
  equals(), 2-26
  hashCode(), 2-26
  reset(), 2-27
Microsoft Internet Information Server, 5-3
mod_jserv, 1-15
monitoring
  resource use, 1-18
mutex, 2-22

N

Noah's Ark, 1-4

O

objects
  examples, 2-26
  factory, 2-2
  immutable, 2-24
  recycling, 2-27
  reuse, 2-26
  unused, 2-28
optimization
  application performance, 1-20
Oracle HTTP Server
  and certificates, 4-17
  tuning, 2-29
Oracle JavaServer Pages, 2-6
Oracle9iAS Database Cache, 2-11
Oracle9iAS Web Cache, 2-11
  and Oracle HTTP Server, 2-11
  and third-party servers, 5-2
overflow
  buffer, 4-8

P

page buffer, 2-18
passwords, 4-10
performance
  effect of HTTPS with SSL, 4-14
  evaluating, 2-33
  goals, 2-33
  JServ, 2-19
  methodology, 2-33
  testing, 2-32, 2-33, 4-15
  tradeoffs, 2-2
personnel
  testing, 2-34
pooling
  connections, 2-9
  definition, 2-9
  JDBC connections, 2-9
POP3, 4-5
preload
  servlets, 2-20
pre-translate
  JSP, 2-20
privileges
  and modules, 4-8
  root, 4-10
procedures
  documenting, 1-28
proxies
  on DMZ, 4-6
  reverse, 4-14

R

recording
  failures, 1-6
redirect, 2-17
redundant components
  and availability, 1-3
  and load balancers, 1-7
reissuing
  certificates, 4-19
requests
  duplicate, 1-23
reset(), 2-27
resource use
  and Dynamic Monitoring Service, 1-18
  application bugs, 1-18
  monitoring, 1-18
restart
  server, 1-25
retry
  transaction, 1-22
reverse proxies, 4-14
root privileges, 4-10
router, 4-3
rules
  caching, 5-5
  expiration, 5-5

S

security patches, 4-10
server
  restart, 1-25
servers
  legacy, 5-2
  placement, 4-5
  virtual, 4-15
ServerVariables_JScript.asp, 5-14
services
  unused, 4-10
servlets
  deployment, 2-18
  preload, 2-20
  sample code example, 2-37
  startup, 2-15
session scope beans
  case study, 2-7
session state, 1-19
sessions
  case study, 2-7
  drawbacks, 2-7
  guidelines, 2-8
  tracking, 2-6
SessionSample, 5-11
SessionServlet, 5-7
SingleThreadModel
  description, 2-13
  examples, 2-14
  implementation, 2-13
SMTP, 4-5
SnoopServlet, 5-5
software
  availability, 1-17
  cryptographic, 4-9
spawning
  threads, 2-13
SQL statement
  tuning, 2-12
SSL
  guidelines, 4-14
SSLSessionCacheTimeout
  tuning, 2-31
Stack, 2-25
startup
  servlets, 2-15
stateful inspection, 4-3
stateless
  applications, 2-5
  defined, 2-6
statement caching
  JDBC, 2-10
statesafe
  definition, 1-17
static include
  description, 2-15
step-up digital right, 4-12
String, 2-24, 2-28
  and Java2 compiler, 2-29
StringBuffer, 2-24, 2-28
  and Java2 compiler, 2-29

character buffer, 2-29
switched connection firewalls, 4-4
synchronization
Java, 1-17

T

TCP/IP parameters
setting, 2-30
testing
performance, 2-32, 2-33, 4-15
personnel, 2-34
Thawte, 4-16
threads
spawning, 2-13
thread-safe
wrapper, 2-24
thread-safe JSPs
declare variables, 2-17
performance, 2-16
tradeoffs
firewall architecture, 4-2
performance, 2-2
transaction
retry, 1-22
transfers
HTTPS, 4-15
TreeMap, 2-25
tuning
database, 2-12
KeepAlive, 2-30
MaxClients, 2-30
Oracle HTTP Server, 2-29
SQL statement, 2-12
SSLSessionCacheTimeout, 2-31

U

unused
objects, 2-28
services, 4-10
upgrades
botched, 1-25
browser, 4-13
URL Wildcard, 4-17

V

Vectors, 2-25
verbose mode, 5-6
Verisign, 4-11, 4-12, 4-13

W

weak encryption, 4-11
Web Cache, 2-11
and third-party servers, 5-2
Web site
configuration, 5-3
wrapper
thread-safe, 2-24