Performance and Tuning Guide

iPlanet™ Application Server

Version 6.5

816-2587-10
February 2002

Contents

Preface

This guide is intended for advanced administrators of iPlanet Application Server. This guide helps you tune iPlanet Application Server for maximum performance and reliability. It is recommended that you backup your configuration files, before changing the configuration settings on iPlanet Application Server.

This chapter describes the contents of iPlanet Application Server *Installation Guide*. It contains the following sections:

- Using the Documentation
- About This Guide
- What You Should Know
- How This Guide is Organized
- Documentation Conventions

Using the Documentation

The following table lists the tasks and concepts that are described in the iPlanet Application Server manuals and *Release Notes*. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate manual.

Note that the printed manuals are also available online in PDF and HTML format, at: http://docs.iplanet.com/docs/manuals/ias.html

For information about	See the following	Shipped with
Late-breaking information about the software and the documentation	*Release Notes*	Available on the iPlanet Web site, at http://docs.iplanet.com.
Installing iPlanet Application Server and its various components (Web Connector plug-in, iPlanet Application Server Administrator), and configuring the sample applications	*Installation Guide*	iPlanet Application Server 6.5
Creating iPlanet Application Server 6.5 applications that follow the open Java standards model (Servlets, EJBs, JSPs, and JDBC), by performing the following tasks: • Creating the presentation and execution layers of an application • Placing discrete pieces of business logic and entities into Enterprise Java Bean (EJB) components • Using JDBC to communicate with databases • Using iterative testing, debugging, and application fine-tuning procedures to generate applications that execute correctly and quickly	*Developer's Guide*	iPlanet Application Server 6.5

For information about	See the following	Shipped with
Administering one or more application servers using iPlanet Application Server Administrator Tool to perform the following tasks: • Monitoring and logging server activity • Implementing security for iPlanet Application Server • Enabling high availability of server resources • Configuring web-connector plugin • Administering database connectivity • Administering transactions • Configuring multiple servers • Administering multiple-server applications • Load balancing servers • Managing distributed data synchronization • Setting up iPlanet Application Server for development	*Administrator's Guide*	iPlanet Application Server 6.5
Migrating your applications to the new iPlanet Application Server 6.5 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with iPlanet Application Server	*Migration Guide*	iPlanet Application Server 6.5
Using the public classes and interfaces, and their methods in the iPlanet Application Server class library to write Java applications	*Server Foundation Class Reference (Java)*	iPlanet Application Server 6.5
Using the public classes and interfaces, and their methods in the iPlanet Application Server class library to write C++ applications	*Server Foundation Class Reference (C++)*	Order separately

About This Guide

This *Performance and Tuning Guide* discusses the various features of iPlanet Application Server and how to tune iPlanet Application Server for maximum performance and reliability.

This manual is intended for system administrators, network administrators, evaluators, application server administrators, web developers, and software developers who want to get an understanding of the various tasks and tools available for tuning the performance of iPlanet Application Server.

What You Should Know

Before you begin, you should already be familiar with the following topics:

- Application Servers
- Client/Server programming model
- Internet and World Wide Web
- Windows NT/2000 or Solaris™ operating systems
- Java programming and J2EE

How This Guide is Organized

This Guide is organized as follows:

Chapter 1, “About iPlanet Application Server”, gives an overview of iPlanet Application Server features, and iPlanet Application Server components.

Chapter 2, “Understanding Tuning and Sizing”, provides an insight into sizing, estimating operational requirements, and tunable parameters.

Chapter 3, “Tuning Your Application”, provides a comprehensive guide to tuning your applications for maximum performance. Java Coding Guidelines and J2EE Programming Guidelines are discussed in this chapter.

Chapter 4, “Tuning iPlanet Application Server”, provides a comprehensive guide to tuning iPlanet Application Server for maximum performance.

Chapter 5, “Tuning the Java Runtime System”, provides an insight into memory tuning and Garbage Collector settings.

Chapter 6, “Tuning the Operating System”, provides information on Solaris tuning parameters.

Chapter 7, “Tuning Database Servers”, discusses various database tuning parameters.

Chapter 8, “General Guidelines for Better Performance”, provides guidelines on better coding practices that can improve the performance of your application.

Chapter 9, “Validating Server Performance”, provides information on monitoring and validating server performance after tuning.

Chapter 10, “Frequently Asked Questions”, is a list of most common question asked regarding iPlanet Application Server tuning.

Documentation Conventions

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except forward slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form: `http://server.domain/path/file.html`, where:

- server is the name of the server where you are running the application.
- domain is your internet domain name.
- path is the directory structure on the server.
- file is an individual filename.

The following table shows the typographic conventions used throughout iPlanet documentation.

Table 1 Typographic Conventions

Typeface	Meaning	Examples
`Monospaced`	The names of files, directories, sample code, and code listings; and HTML tags	Open `Hello.html` file. `<HEAD1>` creates a top level heading.
Italics	Book titles, variables, other code placeholders, words to be emphasized, and words used in the literal sense	See Chapter 8 of the *Performance and Tuning Guide*. Enter your *UserID*. Enter *Login* in the Name field.

Table 1 Typographic Conventions

Typeface	Meaning	Examples
Bold	First appearance of a glossary term in the text	**Templates** are page outlines.

Chapter 1

About iPlanet Application Server

iPlanet™ Application Server provides a reliable, available and scalable web services deployment platform. Application programmers can focus on implementing business logic with well-engineered software components and rely on the services offered by iPlanet Application Server for massive scale deployment.

This chapter includes the following sections:

- iPlanet Application Server Components
- iPlanet Application Server Process Architecture
- Communication Within iPlanet Application Server
- iPlanet Application Server Tools

iPlanet Application Server Components

iPlanet Application Server includes various components that need to interact with each other for a smooth performance. These components can be tuned for optimum performance for both production and development environments.

This section describes the following topics:

- Web Connector Plugin
- Application Server Processes
- Directory Server Components
- Databases

Web Connector Plugin

A dynamically loaded library that plugs into a web server instance. Redirects incoming HTTP requests to the Executive Server (KXS) processes in an application server instance.

Application Server Processes

Executive Server (KXS). Redirects incoming HTTP traffic received from Web Connector Plugin to a Java Server. Manages replication of highly available session and state data between application server instances.

Java Server (KJS). Contains web and EJB containers.

RMI/IIOP Bridge (CXS). Redirects incoming requests from RMI/IIOP clients to the EJB container housed in a Java Engine (KJS).

Administrative Server (KAS). Monitors other application server processes and acts as a server for the administrative and deployment tools.

Directory Server Components

Directory Server Process. Contains user and group information for authentication and authorization. Acts as a distributed store for the application server's registry information.

iPlanet Administration Server. Acts as a contact point for the iPlanet Console dministration tool. This server need not active fo rthe application server to function.

Databases

Oracle, Sybase, Informix and DB2, are some of the databases supported by iPlanet Application Server. Using iPlanet Application Server, you can configure 3rd party JDBC drivers for databases. You can also configure datasources and transaction manager for database drivers.

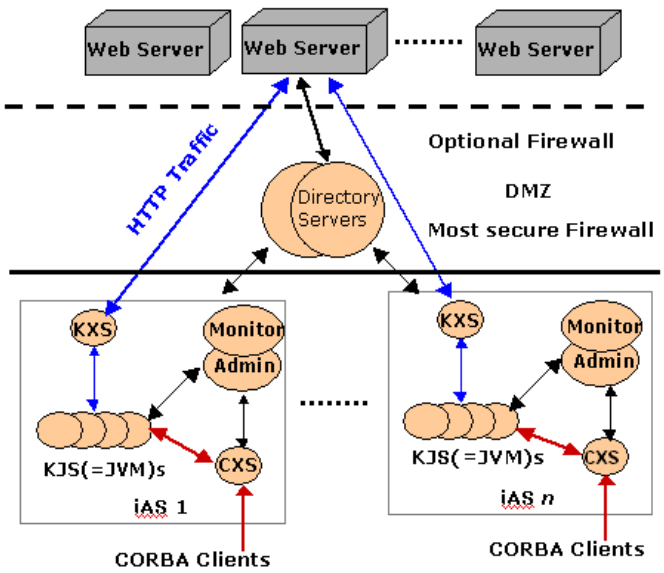
iPlanet Application Server Process Architecture

To achieve maximum performance, you need to understand the basic process architecture of iPlanet Application Server. A common iPlanet Application Server deployment scenario is shown below. For the sake of clarity, the figure depicts only the traffic from one web server instance to iPlanet Application Server and Directory Server. The web server (or more precisely, the Web-Connector Plugin) manages the load balancing of requests to all the nodes configured in iPlanet Application Server cluster, as shown in the figure.

The multi-instance clustering architecture delivers superior horizontal scalability and high availability through replication of state on other application server instance nodes. Each instance can be configured with multiple KJS processes (each KJS process is a JVM instance) to achieve the desired degree of scalability. Each KJS process can be assigned multiple threads for processing, thereby increases performance.

The following illustration will help you understand how all the components of iPlanet Application Server fits together. The illustration shows how the various parts of iPlanet Application Server's process architecture are related to one another and the way they are connected to the web servers, and ultimately, to the users.

Figure 1-1 iPlanet Application Server process architecture



Communication Within iPlanet Application Server

iPlanet Application Servers are typically deployed together with web servers, directory servers, and client database applications. In this scenario, inter-process communication is of a very high level and this makes data integrity and security issues very important. To ensure smooth and secure communication between application servers, web servers and the client applications that run on application servers, security components of the system are frequently separated by firewalls.

The basic component of iPlanet Application Server is the Executive Server (KXS) which creates the components of the application, manages per-session data, load monitoring and load balancing functions with other instances of iPlanet Application Server.

The application code runs in multi-threaded processes created by KXS. There are two types of processes; the C++ Server (KCS) and the Java Server (KJS).

The system is managed through the Administration Server (KAS). The web server may be installed on the same machine as iPlanet Application Server, or on a different machine. In a typical network installation, the web server would be installed on a separate machine.

The communication between the web server and the application server is through a plug-in, which resides in the web server. If multiple instances of the application server have been installed, the plug-in communicates with the application server instance, that the load balancing mechanism selects. In cases where the web server does not have a plug-in available, the communication module could be a CGI application, which establishes connection to iPlanet Application Server. However, the CGI model is less efficient, and is normally used only as a last resort.

The second method of communication is using OCL (Object Constraint Language), which uses CORBA to locate the required services and communicate with them through the application server. This is recommended only for use in an Intranet because standardized security for IIOP connections is not available.

Inter-process communication between servers, applications and security protocol occurs in a number of ways. Although it is possible to place a firewall between iPlanet Application Server instances (an application server instance being a KXS/KJS/KCS process group), this is not recommended. If such a firewall is implemented, then it is necessary to implement IGMP (Internet Group Management Protocol) to allow Internet Protocol (IP) communication across the firewall.

Although firewalls are not recommended between iPlanet Application Server instances, the implementation of a fast, dedicated network for IP Multicast traffic between iPlanet Application Server machines may be advisable in certain circumstances.

The iPlanet Application Server system is designed for networks running at LAN speeds. Distributing iPlanet Application Server instances across a WAN (Wide Area Network) may lead to performance issues. For more information on configuring firewalls with iPlanet Application Server, refer to Chapter 5, “Securing iPlanet Application Server”, in *iPlanet Application Server Administrator’s Guide.*

iPlanet Application Server Tools

The Administrative and Deployment interfaces of iPlanet Application Server are managed using two tools: the iPlanet Application Server Administration Tool (iASAT) and the iPlanet Application Server Deployment Tool (iASDT). These tools are discussed in the following topics

- iPlanet Application Server Administration Tool
- iPlanet Application Server Deployment Tool

iPlanet Application Server Administration Tool

iASAT is a stand-alone Java application with a graphical user interface that allows you to administer one or more instances of the iPlanet Application Server. Administration often involves performance-related tasks such as adjusting database connection threads and changing load-balancing parameters. The administration tool can run on any platform that is supported by iPlanet Application Server and allows you to connect to one or more iPlanet Application Server instances over the network.

Use iPlanet Application Server Administration Tool to tune server processes, EJB parameters, configure datasources, etc., to enhance the application server’s performance. This guide will show you how to use the administration tool to tune the application server.

iPlanet Application Server Deployment Tool

iASDT is a stand-alone Java application with a graphical user interface that allows you to do the following:

- Package J2EE application components into modules
- Assemble the modules into a deployable unit, and
- Deploy the units to one or more iPlanet Application Server operating environments.

J2EE application components are archived into modules according to the container that will receive them upon deployment. You can archive J2EE application components into an EJB JAR module (archived with a `.jar` extension) or a Web Application module (archived with a `.war` extension). Each module will contain a J2EE descriptor and an iPlanet Application Server specific deployment descriptor, in XML files.

Chapter 2

Understanding Tuning and Sizing

This chapter describes performance tuning tips and techniques. It is a concise guide to some of the tunable parameters of iPlanet™ Application Server.

This chapter contains the following topics:

- Why Tune iPlanet Application Server?
- What Is Application Sizing?
- Factors That Affect Sizing
- Understanding Operational Requirements
- Predicting Performance
- General Performance Guidelines
- Performance Tuning Sequence

Why Tune iPlanet Application Server?

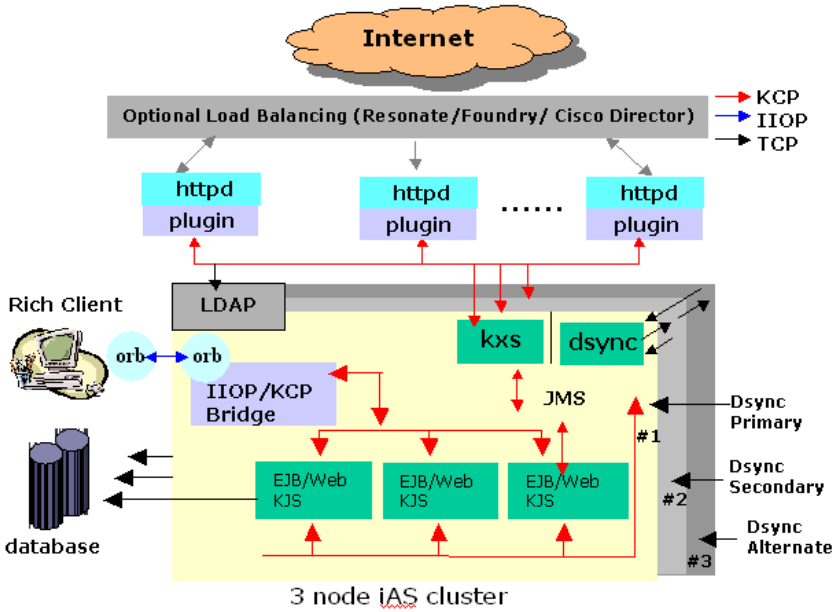
There are so many variables that influence the performance of an application server that there is no single configuration that performs optimally at every site. Even if the system is sized properly, performance can be significantly enhanced by just a few modifications made to the default deployment.

These modifications can enhance the overall system performance. Certain modifications have a positive impact on the development environment, but adversely effect production environments. The reverse is also true.

Before you tune iPlanet Application Server, you need to assess the exact result you are expecting. This chapter takes you through various tuning and sizing options that you can adopt to get the best out of your iPlanet Application Server.

The process architecture of iPlanet Application Server is represented in the following diagram for easy reference:

Figure 2-1 iPlanet Application Server Process Architecture



What Is Application Sizing?

The sizing of a system is meant to forecast the amount and type of hardware necessary to support a particular user load.

To properly size a system, it is important to understand the characteristics of the load, the characteristics of the application, and the characteristics of the platform (operating system and hardware).

This section of the document discusses the details for measuring the load, the application, the platform, and to estimate the system requirements for the application servers.

Before we can move on to the methods used to size an application, we need to understand the factors that affect sizing.

Factors That Affect Sizing

The following table lists the factors that affect the sizing of an application

Table 2-1 Factors that affect sizing

Factor	Description
User load	The system load must be proportionate to the hardware required to handle the load.
Application Design & Implementation	An application that performs very little work will be able to handle a lot of users for a given amount of hardware. Often, this kind of application scales poorly because it spends a large percentage of its time waiting for shared resources. Conversely, applications that perform a great deal of computation tend to require much more hardware per user, but scale much better. An application whose threads contend for shared resources is likely to scale poorly within a single server, spending time waiting for those shared resources. The optimal solution for this would be to employ a number of small servers. An application that contends for shared resources between servers is likely to scale poorly within a cluster, spending time waiting for shared resources. The optimal solution for this would be to employ fewer, large servers.
Hardware Platform	Raw processor performance is critical to reducing the amount of hardware needed. Generally, applications do not include floating point intensive computation, so internal performance is usually the most important factor. Response time is highly dependent on the performance of the processor.
Safety Margins	Even with high-speed processors, a server can scale poorly if shared resources cause significant contention. Usually, cache design and memory bandwidth play a big role in determining how much extra performance is achieved as processors are added to a server.

Understanding Operational Requirements

As you begin to tune simple deployments, the implementation team can establish the general layout of the operational environment based on the requirements of iPlanet Application Server, and your application. This general layout will show the relative position of the system's external interfaces. Before the general layout can be transformed into a more concrete design, implementers need to factor in the following operational requirements of the new system:

- Security
- Availability
- Performance

Security

If you want to encrypt the communication between the browser and the web server, you need to consider the following factors:

- Do you need to encrypt the browser to Web server communications for all or part of the application?
- Will the Web server tier exist in a demilitarized zone (DMZ), separate from the application server tier and backend enterprise systems?
- Is encryption required between the web servers and application servers?

For information on how to encrypt the communication between Web servers and iPlanet Application Server, please refer to Chapter 6, "Enabling High Availability of Server Resources", in *iPlanet Application Server Administrator's Guide.*

Availability

You should consider the following factors to ensure the availability of an application:

- What are the availability requirements of the application? Is the loss of service acceptable when a machine becomes unavailable?
- Can the loss of a user's session information be tolerated?
- What are the possible weak links with respect to the manner in which the application interacts with other aspects of the environment?
- Can these weak links be reasonably enhanced or are they constants?
- Many companies view a CPU busy rate of 80% as a high-water mark. What is your company's standard?

Performance

You should consider the following factors with regard to performance:

- What are the required response times experienced by the end users for various interactions with the application?
- What are the perceived steady state and peak user loads?
- What is the average and peak amount of data transferred per Web request?
- What is the expected growth in user load over the next 12 months?

For peak user loads, you need to focus on the number of concurrent sessions being managed by the application server. We often find that organizations view peak user load as the total number of possible users rather than the average number of concurrent users. Given this more realistic view of user loads, you'll find that the number of peak users drops dramatically on paper from hundreds of thousands or even millions to tens of thousands of concurrent users.

Defining these operational requirements will help move you to the next stage in understanding the deployment environment. Let us assume that the operational security and availability requirements are such that multiple Web and application server instances, separated by a set of firewalls, will form the basis of the environment. Here, you'll need separate tiers of machines to support the division between the DMZ and the secure, back-end business systems. You'll also need to plan on multiple instances of machines in each tier to enhance the availability of the application.

Proceeding from these assumptions, the layout of the operational environment is further refined, though it does not yet address the exact number or size of machines required by the system. The next step is to develop a basic understanding of how to predict performance and how to size a system.

Predicting Performance

Given the factors affecting the sizing process and the general layout of the operational environment, how does one predict either the capacity of a given combination of hardware, or the minimal hardware required to sustain a specified capacity? The best way to answer these questions is to take the data gathered from the discussions in the previous section and plug it into the sizing calculator that is bundled with iPlanet Application Server.

iPlanet provides you with two calculators to help size applications deployed to iPlanet Application Server. The first calculator computes the size of a system (i.e., the number of CPUs and the number of machines in each tier) based on the factors described in the above section. The second calculator computes the maximum capacity of a given hardware configuration.

iPlanet has built these calculators based on a combination of tests, including those for popular application workflows, and has drawn upon publicly available benchmark results for RDBMS and processors. Both calculators assume a fully tuned system.

Apart from using sizing calculators, it is recommended that you develop your own understanding of application sizing based on the following steps:

1. Determine performance on a single CPU

   You need to first determine the largest load that can be sustained with a known amount of processing power. You can obtain this figure by measuring the performance of the application on a uniprocessor machine. You can either leverage the performance numbers of an existing application with similar processing characteristics or, ideally, use the results of basic performance testing done during development.

   While determining performance on a single CPU, you must begin to tune the basic environment. As with any performance test, you must ensure that none of the outlying systems (driver machines, Web servers, database machines, etc.) throttle the test. Otherwise, your performance numbers may be artificially low and will adversely impact the sizing results.

2. Determine vertical scalability

   You need to know exactly how much additional performance is gained when you add processors. That is, you are indirectly measuring the amount of shared resource contention that occurs on the server for this workflow. You can either obtain this information based on additional load testing of the application on a multiprocessor system, or leverage existing information from a similar application that has already been load tested. Running a series of performance tests on one to four CPUs generally provides a decent sense of the vertical scalability characteristics of the system.

   Based on your sizing estimates, it's important to exercise the application under load on systems of the target configuration. While determining the vertical scalability, ensure that availability requirements are factored into the configuration. For example, to guarantee that the failure of a single JVM does not result in a loss of all sessions, perform the vertical scalability tests with at least two JVMs and configure session replication between the JVMs.

3. Determine horizontal scalability

   You need to know how much additional performance is gained when you add servers. Again, benchmarking of a cluster of application server systems is required if information on a similar application is not already available. Ensure that you take into consideration high-availability requirements and the attendant session replication configuration as you lay out your horizontal scalability test environment.

   In this case, session replication occurs across application server instances deployed to multiple machines, in addition to session replication across JVMs within each application server instance. Running this suite of tests will provide you with a solid understanding of the performance of the application server. Using this information, you can develop your own custom sizing equations.

   You need to first determine how much load can be sustained with a known amount of processing power. You can obtain that figure by measuring the performance for a given application on a uniprocessor for extrapolating on multiprocessors.

The equation for sizing a basic (single instance installation) High Availablity (HA) site looks like this:

```
TotalProcessorCount =(P*(1.0-k) * (ks+P*k*K*(1.0-2.0 * K) +sqrt(4.0
*ks*K*(1.0-K)*(ks+P*k*(1.0-2.0*K)))) / ((ks-P*k*K) * (ks-P*K*K));
```

where:

- `P` = Performance (throughput) required at peak load. If a maximum level of CPU usage is specified, then divide `P` by that value. For example, if the servers should never be more than 80% busy for 10,000 users, then `P = 10,000/0.80 = 12,500`.
- `k= (1-CPU scalability) / (1+ CPU scalability)`. CPU scalability is measured by running the application on servers with different numbers of CPUs. In this case, CPU scalability is the additional performance gained by adding a second processor. CPU scalability is dependent on the workflow, the application server software, the operating system, and the server hardware.
- `ks= raw CPU performance`. This is required for each unit of throughput (`P`) for this application on this hardware. `ks` is measured on at least one platform and is extrapolated using the ratio of `SPECint_base95` performance on the new platform, as against the measured platform.

- `K = (1 - server scalability) .` (1+server scalability) - server scalability is measured by running the application on clusters with different numbers of servers (or application server instances). In this case, server scalability is the additional performance gained by adding a second appserver of the same performance as the first.

This equation estimates the total (minimum) number of processors needed. This value assumes that the configuration will be optimized. Note that each of these values is different for different applications. Different equations are used to identify the optimal number of clusters, the number of application server instances and the number of processors per server. This value also assumes that the required peak load must be satisfied with one failed server.

Another method for sizing a single server is to establish a throughput rate required for the site and decide if the workflow fits one of three basic examples given below:

- An online store implemented with servlets. Commonly accessed pages are cached.
- An online store implemented with servlets, but there are no commonly accessed pages. So, caching is disabled.
- An online store implemented with a mix of JSPs, servlets, session and entity beans. Caching is disabled.

General Performance Guidelines

The following table describes the factors that affect sizing and their impact on an application's performance:

Table 2-2 Factors that affect sizing - applying concepts

Concept	Applying the Concept	Measurement	Value Sources
User Load	Peak Concurrent Sessions	Transaction Rate (RPM, WIPS)	Number of Peak Concurrent Users / Period between clicks. Number of Subscribers Session time / Period between sessions.

Application Design and Implementation	Transaction Rate per Measure of CPU performance	RPM per SPECint_base95	Measured from benchmarks
	Scalability within a server (additional performance for additional CPU)	%	% based on curve fitting from benchmarks
	Scalability within a cluster (additional performance for additional server)	%	% based on curve fitting from benchmarks
Hardware platform	Processor performance (usually integer performance)	Ratio of performance of SPARC®20@40Mhz	SPECint_base95
	Scalability (probably cache design & memory bandwidth)	%	% based on curve fitting from benchmarks
Safety Margins	High Availability Requirements.	If yes, size the system assuming that 1 server system is down	Different equations used if High Availability is required.
	% Busy High Water Mark	%	Usually 80%; you need to compute the level of acceptable risk.

Performance Tuning Sequence

We recommend that you tune iPlanet Application Server and its associated elements in the following sequence:

- Tuning Your Application
- Tuning iPlanet Application Server
- Tuning the Java Runtime System
- Tuning the Operating System
- Tuning Database Servers

These topics are discuused in detail in the following chapters.

Chapter 3

Tuning Your Application

This chapter provides a comprehensive guide to tuning your applications for maximum performance. The following topics are discussed in this chapter:

- Java Coding Guidelines
- J2EE Programming Guidelines

Java Coding Guidelines

In this section, we will cover issues related to Java coding and performance related issues. The guidelines laid down in this section are not specific to iPlanet™ Application Server, but are general rules to be followed while coding Java applications:

- Avoid serialization and deserialization

  In Java, serialization and deserialization of objects is a CPU-intensive procedure and is likely to slow down your application.

- Avoid Arrays

  This is a tough recommendation to follow but the use of arrays should be minimized in Java. One of the primary goals of Java is safety, so many of the problems that plague programmers in C and C++ are not repeated in Java. A Java array is guaranteed to be initialized and cannot be accessed outside of its range.

  This range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run-time. While the amount of time used is minimal, it is nevertheless a factor when a large number of arrays are used in code.

- Use `StringBuffer.append()` instead of the "+" operator

In Java, strings are immutable i.e. they never change after creation. For example, the following sequence,

```
String str = "testing";
str = str + "abc";
```

is understood by the compiler as:

```
String str = "testing";
StringBuffer tmp = new StringBuffer(str);
tmp.append("abc");
str = tmp.toString();
```

Thus, copying is inherently expensive and can become a significant factor in hindering performance in case it is overused. We recommend that you use `StringBuffer.append()`.

- Explicitly assign `null` value to de-referenced variables

  Doing this helps the garbage collector easily identify the parts of memory that can be safely reclaimed. Java does not prevent you from using excessive amounts of memory or from cycling through too much memory (e.g., creating and de-referencing many objects). You can get memory leaks by holding on to objects without releasing references. This stops the garbage collector from reclaiming those objects, resulting in increasing amounts of memory being used. Thus, explicitly dereferencing variables by setting them to `null` improves performance.

J2EE Programming Guidelines

The J2EE model defines a framework for application development. It defines the use of JSPs, servlets and EJBs (in addition to JNDI, JMS, and JTS) in application architecture. While all parts of the J2EE model have their uses, some issues need to be kept in mind while designing the architecture. They are:

- Servlet programming guidelines

  All applications in iPlanet Application Server are serviced by JSPs or servlets (which are also entry points to EJBs). In the case of the servlet multithread model (the default model), a single instance of a servlet is created for each JVM. All requests for a servlet on that JVM share the same servlet instance. This can create thread contention. Thus, the use of class variables should be avoided as it creates synchronization problems.

In addition, the use of the synchronization clause (around code or around methods) should be avoided as this creates critical sections of code. Only a single thread can execute in the synchronized block at one time. All others are blocked and have to await access. This wait queue can be a significant factor for high performance websites.

- Avoid the use of EJBs

  EJBs are very useful in the context of reusable services. However, this flexibility comes at a cost. This is because the method in which EJBs are designed to work rather than any quirk in the implementation of a container (in this case, the container will be iPlanet Application Server). Often, in J2EE applications, requests to EJBs are routed through servlets.

  Servlets need to do a JNDI lookup, get a bean reference and use that reference to call a bean method. Normally, the reference is cached and used for all subsequent hits. Due to the levels of direction that need to be executed before an EJB can be accessed, it is found that EJBs are inherently more expensive than servlets that perform the same task.

  If you need to use EJBs, please follow these steps to improve response time:

  - Cache EJB references at the servlet. This will avoid the need to do a JNDI lookup for every request.
  - The following EJB types are listed in descending order, based on their performance. The bean-type with the highest performance is at the top:
    - Stateless Session Beans
    - Stateful Session Beans
    - Entity Beans with CMP (Container Managed Persistence)
    - Entity Beans with BMP (Bean Managed Persistence)

    Stateless session beans are the fastest among EJBs and are almost comparable to servlets in performance.
  - Use sticky load balancing in the case of stateful session beans. If sticky load balancing is not used and a bean reference (that has been stored in session) gets routed to another iPlanet Application Server, then a cross container lookup will need to be done to service a request. This can be very expensive.
  - Size the container (iPlanet Application Server) based on performance tests. Configure threads for processes and set time-out values for iPlanet Application Server. Configure EJB cache for improved performance.

- Try to deploy servlets and JSPs to iPlanet Application Server rather than to the iPlanet Web Server. Deploy an application to iPlanet Application Server if:
  - An application is highly transactional
  - Requires failover support to preserve session data,
  - Accesses legacy data.
- Deploying an application to iPlanet Web Server is useful if an application is mostly stateless, read-only, and non-transactional.
- Ask the server administrator to co-locate EJBs with your presentation logic (servlets and JSPs) on the same server to reduce the number of Remote Procedure Calls (RPCs) when the application runs.

NOTE Decomposing an application into a moderate to large number of separate EJBs can create a huge application performance degradation and more overhead. EJBs, like JavaBeans, are not simply Java objects. EJBs are higher level entities than Java objects. They are components with remote call interface semantics, security semantics, transaction semantics, and properties.

Chapter 4

Tuning iPlanet Application Server

This chapters provides a comprehensive guide to tuning iPlanet™ Application Server for maximum performance. The following topics are discussed in this section:

- Optimizing Performance of Server Processes
- Comparing Distributed and Lite HTTP Sessions
- Configuring a Single Backup for Highly Available Sessions
- Configuring Dsync Session Management Threads
- Load Balancing Options
- Configuring Database Connection Pool
- Configuring EJB Parameters For Runtime
- Caching JSPs and Servlets

Optimizing Performance of Server Processes

The Executive Server (KXS), java engine (KJS), C++ engine (KCS), and RMI/IIOP bridge process (CXS) form the core of iPlanet Application Server. In this section, we will discuss how to tune these processes for maximum performance and scalability.

This section describes the following topics:

- Tuning iPlanet Application Server Processes
- Performance Tuning RMI/IIOP

Tuning iPlanet Application Server Processes

The Executive Server (KXS), the java engine (KJS) and C++ engine (KCS) process requests asynchronously by employing a pool of worker threads. These threads handle user requests for application components. When iPlanet Application Server receives a request, it assigns the request to a free thread. The thread manages the system needs of the request. For example, if the request needs to use a system resource that is currently busy, the thread waits until that resource is free before allowing the request to use that resource.

You can adjust the number of request threads globally, for all processes used by that instance of iPlanet Application Server. You can also do this at process level.

NOTE Note that the process level setting overrides the server level setting. You can tune these settings using the iPlanet Application Server Administration Tool.

The following topics are discussed in this section:

- Optimizing KXS Performance
- Optimizing KJS Performance
- Adjusting the Number of Request Threads
- Specifying Maximum Server and Engine Shutdown Time

Optimizing KXS Performance

The Web Connector Plug-in routes users requests aimed at iPlanet Application Server applications, to the Executive process (KXS). These requests are logged to the request queue in the Executive process.

You can perform the following tasks to optimize KXS performance:

- Control the maximum number of threads the Web Connector Plug-in will use to process requests. This prevents the request queue from receiving more requests than it can process. On iPlanet Application Server installations, the number of KXS threads is set to 32 by default. This can be increased all the way to 128. A setting of 64 threads is sufficient. If the thread count is to be increased, it is recommended that KXS be bound to at least a single processor to avoid wasted time in threads acquiring mutex locks.

- Set the maximum number of requests that are logged to the request queue to control the flow of requests. The maximum number is called the “high watermark”.
- Set the number of requests in the queue at which logging will resume. This number is called the “low watermark”.
- Bind KXS to a single processor or a processor set. This should only be done if requests are queued at KXS when testing loads. This should not be done for KJS because in iPlanet Application Server, the JDK is optimized for multiple processors and binding it to a single processor does not buy any performance benefits. In addition, if binding to a single processor does not improve KXS performance (as seen by CPU utilization being high on the KXS process), then you can create a processor set using two processors. KXS should then be bound to the processor set.

When a server process, such as Executive Server (KXS), Java Server (KJS), C++ Server (KCS) or Corba Executive Server (CXS) fails, the Administrative Server restarts it. You can set the restart option to either increase or decrease the number of times that a process is restarted. Fault tolerance and application availability are increased when all processes are running smoothly.

Optimizing KJS Performance

When you install iPlanet Application Server, the number of KJS threads is set to 32. This can be increased all the way to 48. A setting of 48 KJS threads is optimal.

Adjusting the Number of Request Threads

The thread pool is by default populated with 8 threads in each process. The maximum is set to 32 threads. You can specify the minimum and maximum number of threads that are reserved for requests from applications. The thread pool is dynamically adjusted between these two values. The minimum thread value you specify holds at least that many threads in reserve for application requests. That number is increased up to the maximum thread value that you specify.

Increasing the number of threads available to a process allows the process to respond to more application requests simultaneously. You can add and adjust threads for each process, or you can define the number of threads for all processes under a server, at the server level.

The optimal setting for these parameters would vary based on the application. For example, if the request involves significant amount of database processing and the database server hardware is lightly loaded and can handle increased concurrency, it is advisable to tweak the pool size up and allow greater number of requests to

reach the database and improve throughput. In general, a larger thread pool size appears to benefit, till about 32-48 threads in both KXS and KJS processes. We recommend that this be fixed, by setting the minimum and maximum pool size to the same number, initially at 32, and, if necessary experiment with 48 threads. You can specify all the required parameters in one go, using iASAT.

By default, each process uses the threads assigned to iPlanet Application Server. For example, if iPlanet Application Server uses a minimum of 8 threads and a maximum of 64 threads, each individual process uses a minimum of 8 threads and a maximum of 64 threads.

Specifying Maximum Server and Engine Shutdown Time

You can set the maximum number of engine restarts of the Administration Server for both iPlanet Application Server and engine processes. For example, if you set the engine shutdown time to 60 seconds, application tasks being processed are allowed 60 seconds for completion. No new requests are accepted after this period has elapsed. Specifying a shutdown value avoids a “hard” shutdown that will return errors to the client. You can set these values using iPlanet Application Server Administration Tool.

Maximum Server Shutdown Time. The Maximum Server Shutdown Time is the maximum time taken to shut down iPlanet Application Server. After this time, any engines that are still running are killed. The server typically shuts down quickly unless it is heavily loaded.

Maximum Engine Shutdown Time. The Maximum Engine Shutdown Time is the maximum time that iPlanet Application Server will wait for an engine to shut down. After this time, the engine will be killed, and the next engine(s) will be shutdown.

Switch off all Logging

To reduce the strain on the system owing to continuous input / output operations, switch of all application logging that will be written to the KJS logs. This action has a marked improvement on performance.

Set MaxBackups = 1

In a normal application server architecture, a single Sync Backup server will reduce the amount of intra-cluster communication. For a cluster, set `Maxbackups` (maximum number of backups) to 1. Setting it to 0 will mean that there are no session backups in case the primary becomes unavailable. Setting it to 2 will increase intra-cluster communication, increasing the load on the server. Therefore, a setting of 1 is optimal for this parameter.

Performance Tuning RMI/IIOP

For deployment environments in which you expect the RMI/IIOP path to support more than a handful of concurrent users, you should experiment with the tuning guidelines described in this section. The default configuration of the JVM and the underlying OS do not yield optimal performance and capacity when you are using RMI/IIOP.

This section covers the following topics:

- Recognizing Performance Issues
- Basic Tuning Approaches
- Enhancing Scalability
- Firewall Configuration for RMI/IIOP

Recognizing Performance Issues

Before exercising your RMI/IIOP client application under load, ensure you have verified that basic mechanical tests are completed successfully.

As you begin exercising the client application under load, you may experience the following exceptions on the RMI/IIOP client:

```
org.omg.CORBA.COMM_FAILURE
```

```
java.lang.OutOfMemoryError
```

```
java.rmi.UnmarshalException
```

If you've verified that the basic mechanics of your application are working properly, and you experience any one of these exceptions while load testing your application, see the next section to learn how to tune the RMI/IIOP environment.

Basic Tuning Approaches

You should experiment with the following tuning recommendations in order to find the best balance for your specific environment.

Solaris File Descriptor Setting. On Solaris, setting the maximum number of open files property using ulimit has the biggest impact on your efforts to support the maximum number of RMI/IIOP clients. The default value for this property is 64 or 1024 depending on whether you are running Solaris 2.6 or Solaris 8. To increase the hard limit, add the following command to /etc/system and reboot once:

```
set rlim_fd_max = 8192
```

You can verify this hard limit by using the following command:

```
ulimit -a -H
```

Once the above hard limit is set, you can increase the value of this property explicitly (up to this limit) using the following command:

```
ulimit -n 8192
```

You can verify this limit by using the following command:

```
ulimit -a
```

For example, with the default ulimit of `64`, a simple test driver can support only `25` concurrent clients, but with ulimit set to `8192`, the same test driver can support `120` concurrent clients. The test driver spawns multiple threads, each of which performs a JNDI lookup and repeatedly calls the same business method with a think (delay) time of 500ms between business method calls, exchanging data of about 100KB.

These settings apply to both RMI/IIOP clients (on Solaris) and to the RMI/IIOP Bridge installed on a Solaris system. Refer to Solaris documentation for more information on setting the file descriptor limits.

Java Heap Settings. Apart from tuning file descriptor capacities, you may want to experiment with different heap settings for both the client and Bridge JVMs. For more information, see Chapter 5, “Tuning the Java Runtime System”.

Enhancing Scalability

Beyond tuning the capacity of a single Bridge process and client systems, you can improve the scalability of the RMI/IIOP environment by using multiple RMI/IIOP Bridge processes. You may find that configuring multiple Bridge processes on the same application server instance improves the scalability of your application deployment. In certain cases, you may want to use a number of application server instances each configured with one or more Bridge processes.

In configurations where more than one Bridge process is active, you can partition the client load by either statically mapping sets of clients to different Bridges or by implementing your own logic on the client side to load balance against the known Bridge processes.

Firewall Configuration for RMI/IIOP

If the RMI/IIOP client is communicating through a firewall to the iPlanet Application Server, you must enable access from the client system to the IIOP port used by the RMI/IIOP Bridge processes. Since the clients port numbers are assigned dynamically, you must open up a range of source ports and a single destination port to allow RMI/IIOP traffic to flow from a client system through a firewall to an instance of the application server.

A snoop-based trace of the IIOP traffic between two systems during a single execution of the Converter sample application is given below. The host `swatch` is the RMI/IIOP client, while the host `Mamba` is the destination or application server system. The port number assigned to the RMI/IIOP Bridge process is 9010. Note that the two dynamically assigned ports (`33046` and `33048`) are consumed on the RMI/IIOP client, while only port `9010` is used to communicate with the Bridge process:

```
swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Syn
Seq=140303570 Len=0 Win=24820

Options=<nop,nop,sackOK,mss 1460>

mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Syn
Ack=140303571 Seq=1229729413 Len=0 Win=8760

Options=<mss 1460>

swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729414
Seq=140303571 Len=0 Win=24820

swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729414
Seq=140303571 Len=236 Win=24820

mamba.red.iplanet.com -> swatch TCP D=33046 S=9010 Ack=140303807
Seq=1229729414 Len=168 Win=8524

swatch -> mamba.red.iplanet.com TCP D=9010 S=33046 Ack=1229729582
Seq=140303807 Len=0 Win=24820

swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Syn
Seq=140990388 Len=0 Win=24820

Options=<nop,nop,sackOK,mss 1460>

mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Syn
Ack=140990389 Seq=1229731472 Len=0 Win=8760

Options=<mss 1460>

swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731473
Seq=140990389 Len=0 Win=24820
```

```
swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731473
Seq=140990389 Len=285 Win=24820

mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990674
Seq=1229731473 Len=184 Win=8475

swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731657
Seq=140990674 Len=0 Win=24820

swatch -> mamba.red.iplanet.com TCP D=9010 S=33048 Ack=1229731657
Seq=140990674 Len=132 Win=24820

mamba.red.iplanet.com -> swatch TCP D=33048 S=9010 Ack=140990806
Seq=1229731657 Len=25 Win=8343
```

Comparing Distributed and Lite HTTP Sessions

Distributed Sessions offer improved availability by replicating session data on a backup iPlanet Application Server node. To acheive this, objects placed into a Distributed Session must implement the `java.lang.Serializable` interface.

However, this generates additional network traffic if the session sizes are large and are written to quite often. Due to queuing effects, JSPs and servlets that access distributed sessions also suffer a performance loss, the magnitude of which depends on the amount of `HttpSession` usage and size. Lite Sessions trade-off availability for better performance. Session Objects are locally cached in the KJS process, which serves as a home for all the requests in that session.

Lite Sessions also have the advantage when Java objects are stored in `HttpSession`. Since Lite session objects are cached in each KJS process, there is no need to serialize Java objects over the network. So if a Java object is stored in a Lite HttpSession, there is no need for that object to implement the `java.lang.Serializable` interface.

You can specify the desired session type in `ias-web.xml`, by editing the `<session-info>/<impl>` property.

NOTE Gains of 8-40% have been reported by switching to Lite sessions.

Configuring a Single Backup for Highly Available Sessions

This only applies if you are using the highly available sessions (dsync-distributed) and have configured a primary and secondary for `HttpSession` failover. The registry setting `Maxbackups` for the cluster should be set to `1`. Setting it to `0` will mean that there are no session backups in case the primary becomes unavailable. Setting it to `2` will increase intra cluster chatter. A setting of `1` is optimal for this parameter (and is the default).

You can set this parameter in the following iPlanet Registry key:

```
Software\iPlanet\Application
Server\6.0\Clusters\<cluster-name>\MaxBackups
```

Configuring Dsync Session Management Threads

Dsync is a distributed state synchronization service. The designated Dsync primary and Backup contain in-memory database of session nodes. As Http Sessions time-out, or are invalidated, they need to be promptly removed from the memory database. This removal and book keeping can be done with greater concurrency by increasing the number of dedicated threads.

Set the following property in the registry. This can solve transient memory growth problems and make the session node invalidation more efficient. The following sets the KXS cleaner thread count to 20, in iPlanet Registry:

```
Software\iPlanet\Application
Server\6.0\CCSO\ENG\0\SyncTimeoutThreadCount=20
```

To accomplish the same for KJS engine #1, modify the following key in iPlanet Registry:

```
Software\iPlanet\Application
Server\6.0\CCSO\ENG\1\SyncTimeoutThreadCount=20
```

There is another setting to configure the interval at which these threads are activated:

```
Software\iPlanet\Application
Server\6.0\CCSO\ENG\<n>\SyncTimerInterval
```

This defaults to 60 and this setting is quite adequate.

NOTE While this does not improve performance directly, keeping a small session object store always helps. It also ensures that the KXS and KJS processes to not leak memory, by cleaning up promptly and efficiently.

We recommend that you not rely on `HttpSession` time-outs to clean up sessions. Use `HttpSession.invalidate()` method to clean up and where that is not possible, set the default `HttpSession` time-out to be as low as possible in the deployment environment.

Load Balancing Options

This section describes the following topics:

- Load-Balancing Cluster Configuration
- Broadcasting and Updating Information
- Monitoring Load-Balancing Information
- Recommended Load-Balancing Configuration for Clusters
- Optimizing Session Size for Clusters
- Load Balancing Individual JSPs
- Using Sticky Session Load Balancing

Load-Balancing Cluster Configuration

In the iPlanet Application Server environment, a cluster is defined as a collection of servers that shares the responsibility for saving state & session information, performed by the Data Synchronization Service (Dsync).

Dsync, therefore, is the shared resource that constrains the size of any given cluster. As a general rule, each cluster should run with no more than 4 instances of iPlanet Application Server. If more servers are required, then use sticky load balancing from the web server tier to multiple clusters. If state and session information is not stored in Dsync, there is no limit on the number of servers that can be run in parallel. Use the iPlanet Application Server Sizing Tool to find an optimal configuration.

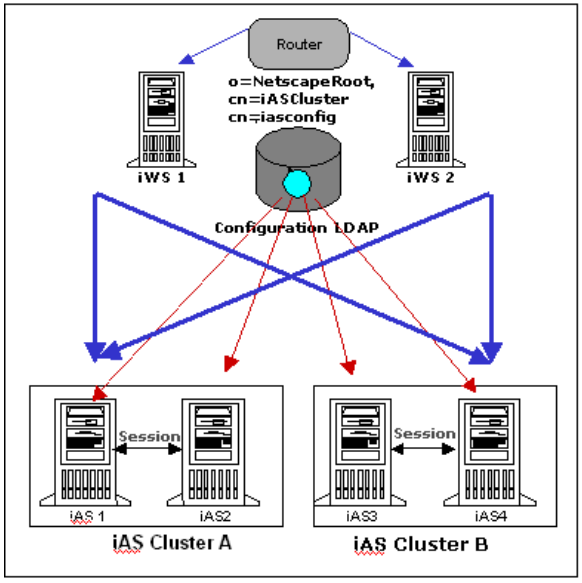
We can explore 2 scenarios for load balancing cluster configuration, as follows:

Scenario 1: Two iPlanet Application Server clusters sharing a single LDAP configuration tree.

In this scenario, all iPlanet Application Servers share the same LDAP configuration tree. To support sticky load balancing, it is not necessary to turn the sticky load balancing method on the router.

The following figure illustrates this configuration. In the figure, incoming requests are represented by sharing configuration LDAP between blue arrows and LDAP access is represented by red arrows.

Figure 4-1 Two Application servers sharing a single LDAP tree



Recommended Usage:

This configuration is useful when each application only exists in a single cluster. Benefits of such a scenario include isolation of applications, and simplicity of the web tier.

Configuration:

During installation of iPlanet Application Server, specify the same LDAP and the same configuration root for all iPlanet Application Server in all clusters.

Scenario 2:

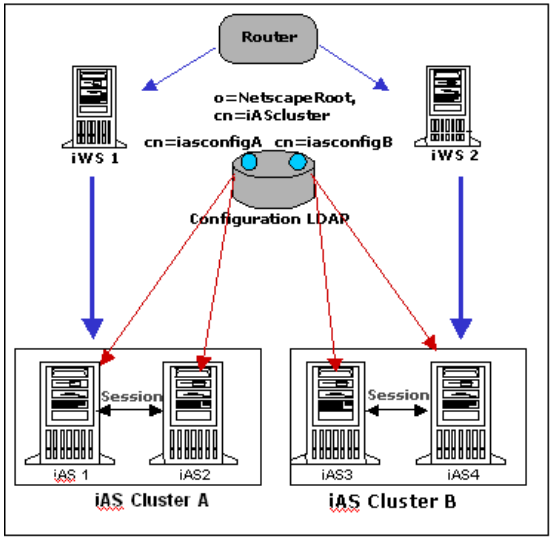
iPlanet Application Server clusters assigned to different LDAP configuration trees:

In this scenario, each iPlanet Application Server cluster has a branch in the configuration tree in LDAP. Each iPlanet Application Server cluster will have at least one iPlanet Web Server dedicated to it. When a request comes in to a iPlanet Web Server (since iPlanet Web Server only knows about one iPlanet Application Server cluster), it will always send requests to iPlanet Application Server in this cluster. The router is responsible for distributing the load between different iPlanet Web Servers.

To enable the support for sticky load balancing, the router's sticky option needs to be turned on so that subsequent requests will come to the same iPlanet Web Server.

The following figure illustrates this configuration. In the figure, incoming requests are represented by sharing the configuration LDAP between blue arrows and user LDAP access is represented by red arrows.

Figure 4-2 Two application servers clusters assigned to separate LDAP tress



Recommended Usage:

This configuration is useful when applications require cross-cluster deployment. Benefits of such a scenario include the ability to use one cluster to promote new versions of applications, and to support applications that require more processing power than a single cluster can provide.

Broadcasting and Updating Information

For load balancing to be effective, each server involved in the process must have the most current information about all the other servers. This means that information about the factors that affect load balancing must be broadcast to all the iPlanet Application Server machines, and every iPlanet Application Server machine must monitor and update this information to make load-balancing decisions. Broadcasting information too often, results in a high level of network traffic and could slow down response time. However, if the load-balancing information is not calculated and updated frequently, then application components risk not being optimally load balanced because the information iPlanet Application Server uses to make load-balancing decisions is outdated.

When making decisions about load balancing, you face two major dilemmas:

- How frequently should an iPlanet Application Server server update its load-balancing information?
- How frequently should every iPlanet Application Server installation broadcast its load-balancing information?

Update Interval. A minimum value of 5 seconds and a maximum value of 10 seconds is appropriate in most cases. In general, set the Update Intervals criteria for each server to be twice the response time, under stable conditions, of the most frequently used application component. For example, on a system where the most frequently used application component returns requests in 5 seconds, set the update interval to 10 seconds. Setting it to a more frequent update rate causes the server to do more work and could even alter load-balancing characteristics. Use caution with this calculation: if the response time of a heavily used application component is only 1.5 seconds, do not set the Update Interval to 3 seconds.

NOTE If the response time of a heavily used application component is only 1.5 seconds, do not set the Update Interval to 3 seconds.

Broadcast Interval. As mentioned earlier, broadcasting load-balancing information too frequently will not only increase network traffic, it will also increase the work load of your iPlanet Application Server as all the servers work to post and gather the information. In general, set the Broadcast Intervals criteria for a server to be twice the value of its Update Interval.

Set the Update Interval and the Broadcast Interval criteria using the Load Balancing tool in iPlanet Application Server Administration Tool.

Monitoring Load-Balancing Information

When you set load-balancing criteria, be patient about the fine-tuning process. Determining the best combination of load balancing criteria takes careful monitoring of your iPlanet Application Server configuration over a period of time, during which you must gather statistics about peak load, your mix of request types, response time averages, bottlenecks, and so on. There is no single load balancing solution for all iPlanet Application Server users, since every system is deployed with different parameters and criteria. As with any aspect of iPlanet Application Server deployment, only you can determine over time the best set of criteria for improving performance of the iPlanet Application Server system deployed at your site.

For more information about load balancing and using iASAT to set load-balancing criteria, see “Balancing User Request Loads” in *iPlanet Application Server Administrator’s Guide.*

Recommended Load-Balancing Configuration for Clusters

During iPlanet Application Server installation, specify the same LDAP for all iPlanet Application Server in all clusters. Specify the same configuration root for all iPlanet Application Server in one cluster and different configuration root for different clusters.

Optimizing Session Size for Clusters

Session size, by far, has the largest effect in the performance of an iPlanet Application Server cluster. After observing large installations of iPlanet Application Server, we have determined that for maximum performance benefit, the session size should not be more than 4K. With larger sessions, the system will continue to work but with degraded performance. The main reason for performance degradation is the constant communication between the primary and the hot backups in the system to synchronize session data.

One of the following techniques can be employed in improving session performance:

- Store only the most important elements in the session.

  The application architect needs to determine data that is important and store only that in the session. Data that need not be distributed should be kept away from the session.

- Use sticky load balancing for sessions

  Sticky load balancing and session distribution are two separate but linked variables in the same equation and both can be enabled at the same time. With sticky load balancing, a single client is always directed to the same KJS. This enables data storage in the JVM memory rather than in the session. Only a key (for example, to a hashtable or to an array) needs to be stored in a session. As the amount of data in a session is significantly reduced, there is an improvement in performance.

  In this configuration, if the iPlanet Application Server which is servicing a client request becomes unavailable, then the request will be routed to another iPlanet Application Server instance in the cluster. Since the key is available in the session, the data can be recreated and stored in memory. The request will now be stuck to the new KJS.

  This option is useful if data in session has been accessed and stored from a secondary source such as an LDAP server.

- Use a separate data store to serialize large session data.

  This concept introduces a new variable into the system - a database. The idea here is to store a large portion of the session in a database and to store only the primary key in session. The session therefore, is smaller and this improves performance. This will involve careful planning of the database schema and proper indexing of the lookups involved to speed up database access.

Load Balancing Individual JSPs

In iPlanet Application Server, JSPs can be load balanced individually. This is done by assigning a GUID to a JSP, similar to how GUIDs are assigned to servlets, in the XML descriptor. (See section on Registered JSPs). By assigning a GUID to a JSP, it becomes possible to load-balance JSPs just as servlets would, through iPlanet Application Server Administration Tool.

Using Sticky Session Load Balancing

The best web performance is achieved when the servlets in a Web application are configured for Sticky load balancing. In this setting, the application server or the Web-Connector Plugin load balance user sessions. The first user request in a session is load balanced to the best candidate KJS process or server. From then on, the same user's subsequent sessions are sent to the same process. This allows the attached KJS process to locally cache the session object and offer better performance. Sticky Load Balancing can be used along with Distributed Sessions.

A servlet or JSP can be configured to be sticky, by setting the `<servlet>/<servlet-info>/<sticky>` property to true in `ias-web.xml`. The deployment and packaging tools automatically set sticky to true.

NOTE Performance improvements of 10% or better have been reported in some tests.

Simplify Session Data

It is better not to serialize an object before storing it in session, for the following reasons:

- The iPlanet Application Server distributed session has been tuned to work with simple data elements. Large serialized objects are clumsy and have a marked effect on performance. The data needs to be stored as separate simple data elements for maximum utilization of the session.

From a Java coding perspective, serialization and deserialization are expensive operations that should be avoided.

Configuring Database Connection Pool

iPlanet Application Server offers a connection pooling feature, which multiplexes a few database connection amongst many threads. Connections are tentatively established for the first use, but are not closed. The already open connections are reused as long as the application server process is alive.

A connection pool will be created for a unique combination of JDBC datasource and user. For the same datasource, multiple pools may be created if the application uses the `DataSource.getConnection` (user,password) style of getting connections where user and password changes with different call invocations.

Make sure the size of this pool is slightly greater than the number of worker threads configured in KJS. Set the `CacheInitSlots` and `CacheMaxConn` properties in the registry for the desired database.

For example, if you are using the bundled native Oracle JDBC Drivers in iPlanet Application Server, modify the properties under the following key in iPlanet Registry.

```
Software\iPlanet\Application Server\6.0\CCS0\DAE2\ORACLE_OCI
```

NOTE The number of connections in pool must be equal to the number of worker threads in KJS process, if you expect that all threads would be concurrently processing user requests and would need database access. The performance benefits will be obvious in a database access intensive application.

This section features the following topics:

- Guidelines for Configuring Connection Pool
- Using Statistics to Configure the Connection Pool

Guidelines for Configuring Connection Pool

Connection Pools in iPlanet Application Server can be configured at various levels:

- In the datasource XML file.
- By modifying the registry.
- Dynamic configuration through the Administration tool.

For more information on the configurable connection pool parameters, see *iPlanet Application Server Administrator's Guide.*

Follow these common guidelines while configuring connection pool:

- For every database backend use one logical datasource. If multiple logical datasources are pointing to the same backend, your resources may not be optimally utilized.
- Keep the reclaim time as high as possible, as it is mainly aimed at reclaiming the connections which are given to applications which never release the connection. ******As there is a side effect of reclaiming the connections even if it is in use after the reclaim time.
- Keep the `maxPoolSize` to the number of physical connections that you wanted to make from this datasource.
- Keep the `minPoolSize` to the average number of concurrent client requests which involve database access to this datasource.
- Before a connection is given to the application from the pool, it will be checked for its sanity. The first one is simple sanity, which is based on `setAutoCommit`, and the other one is table based sanity.

  In most cases, simple sanity will be good enough. However, with certain JDBC drivers it is not possible to recognize stale connections with simple sanity.

  Therefore, use simple sanity only in cases where the database drivers support it and make `isSanityRequired` to `false` if your database backend is reasonably failsafe.
- If the application calls `DataSource.getConnection` (username,password) and the username and password are different each time, then keep the connection pool configuration very low.

Using Statistics to Configure the Connection Pool

iPlanet Application Server supports a rich set of connection pool statistics, which can be used to configure the connection pool.

For more information on how to setup and collect statistics, see *iPlanet Application Server Administrator's Guide.*

Some of the general suggestions are:

- If the Total Connections dropped is not zero, and if the Peek Value for Total Connections in the Pool has not reached `MaxPoolSize`, the database backend will not able to give extra connections.

  To avoid this problem, the database backend can be configured for more number of connections or, the `MaxPoolSize` can be increased.

- If the Peek Value for Queue Size is not zero, then the number of connection requests are more than the `maxPoolSize` and connection requests will get queued. If the Peak Value for ******Queue Size of more than 5 it is advisable to increase the maxPoolSize.

- If the number of Cache Misses are more than zero and if the Peek Value for Total Connections in Pool has not reached `maxPoolSize`, then `minPoolSize` can be increased to a higher value.

Configuring EJB Parameters For Runtime

iPlanet Application Server provides an EJB container that enables you to build distributed applications using your own EJB components, and components from other vendors. When you configure iPlanet Application Server for your enterprise, you must set the EJB container's declarative parameters. These parameters determine, for example, session time-out when an EJB is removed after being inactive for a specified number of seconds. Set these parameters using the iPlanet Application Server Administration Tool.

You can set the following values:

- Default Session Time-out

  Default Session Time-out is `14400` seconds. This denotes the time for which the server can keep a `HttpSession` object alive, before removing it due to inactivity. Set this to an acceptable and much lower value. This applies to stateful session EJBs.

- Default Passivation Time-out

  Default Passivation Time-out is 60 seconds. If the bean creation rate is very low and bean size is large, there may be a need to increase this. However, increasing this value may not impact performance in most scenarios.

  Beans are passivated to the file system and if you see excessive file system activity, there may be excessive passivation activity and possible benefit from tweaking this parameter. This value must be less than the session time-out value.

- Metadata Cache Size

  Meta Data Cache Size is 10 beans. This is a cache of Home Bean handles. You can make this as large as the number of different types of beans that exist in your application. Setting it to 50 or 60 should cover most user applications. Because it caches `EJBHome` instances, subsequent lookups of the same Home interface, will just pick up from cache.

- Implementation Cache Size

  Implementation Cache Size is set at 10 instances. If you expect that N concurrent user sessions to access a stateful or session bean, make sure this is set to equal or larger than N. For stateless session beans, there is perhaps no benefit in setting this larger than the number of KJS threads. The same applies to Entity beans. This iPlanet Application Server Administration Tool setting applies to all deployed beans and is thus too coarse a control.

  The maximum cache size is in number of EJBs.

- Timer Interval

  Timer Interval specifies the interval at which Bean implementation pools are scanned to find candidates for passivation.

  You can also specify this interval in the iPlanet Registry key `CCSO\EB\EbInterval`.

  This parameter determines the entity and stateful session bean clean up interval. Default value is 10 seconds. It was found under experimental conditions that setting the Timer Interval to a lower value leads to frequent pauses in EJB Container, thus affecting the response times. Setting this to a very high value may lead to increased passivation times.

- Failover Save Interval

  Failover Save Interval specifies the time interval at which all active stateful session beans, configured for failover, have their state serialized and passivated to the Dsync in-memory database. This is an expensive operation and can impact performance if the bean size is too high or the save interl is too short.

  See *iPlanet Application Server Developer's Guide*, for guidelines on how to configure and use Stateful session Bean failover support.

  If the server fails, the last saved state of the EJB can be restored. Data saved is available to all engines in a cluster. This value is set on a per server basis and applies to EJBs that were deployed with Failover option enabled (on the General tab of the Deployment Tool EJB descriptor editor).

Caching JSPs and Servlets

You can specify the number of JSP pages that are cached by each KJS engine for each iPlanet Application Server instance. Caching JSPs optimizes application response time.

The cache size is set on a per-page bases. JSP caching aids in the development of compositional JSPs. This provides the functionality to cache JSPs within the java engine, thereby making it possible to have a master JSP which includes multiple JSPs, each of which can be cached using different cache criteria. For example, think of a portal page, which contains a window to view stock quotes, another to view weather information, and so on. The stock quote window can be cached for 10 minutes, and the weather report window for 30 minutes, and so on.

Note that caching of JSPs is in addition to result caching, so its possible that a JSP can be composed of several included JSPs, each of which has a separate cache criterion. The composed JSP itself can be cached in the KXS using the result-caching that becomes available as JSPs now have GUIDs (see section on Registered JSPs in documentation).

Caching of JSPs uses the custom tag library support provided by JSP 1.1. A typical cache-able JSP page looks as follows:

```
<%@ taglib prefix="ias" uri="CacheLib.tld"%>
<ias:cache>
<ias:criteria timeout="30">
<ias:check class="com.iplanet.server.servlet.test.Checker"/>
<ias:param name="y" value="*" scope="request"/>
</ias:criteria>
</ias:cache>
<%! int i=0; %>
<html>
<body>
<h2>Hello there</h2>
I should be cached.
No? <b><%= i++ %></b>
</body>
</html>
```

The <ias:cache> and </ias:cache> delimit the cache constraints. The <ias:criteria > tag specifies the time-out value, and encloses different cache criteria. Cache criteria can be expressed using any or both of the tags, <ias:check> and <ias:param>. The syntax for these tags is as follows:

<ias:criteria timeout="val" > specifies the timeout for the cached element, in seconds. The cache criteria are specified within this and the closing </ias:criteria> <ias:check class="classname" /> This is one of the mechanisms of specifying a cache criteria. The classname refers to a class that has a method called "check", which has the following signature:

```
public Boolean check(HttpServletRequest req)
```

This returns a boolean indicating whether the element is to be cached or not.

<ias:param name="paramName" value="paramValue" scope="request" /> : This is another mechanism to specify cache criteria.

paramName is the name of an attribute, passed in either in the request object (using setAttribute), or in the URI. This is the parameter used as a cache criterion. paramValue is the value of the parameter, which determines whether caching should be performed or not. This can be of the following kinds:

Constraint

Meaning

x = ""

x must be present either as a parameter or as an attribute.

x = "v1|...|vk", where vi might be "*"

The constraint is true of the current request if the request parameter for x has the same value as was used to store the cached buffer.

x = "1-u", where 1 and u are integers.

x is mapped to a value in the range [1, u]

The scope identifies the source of the attributes that are checked. These can be page, request (default), session, or application.

The following is an example of a cached JSP page:

```
<%@ taglib prefix="ias" uri="CacheLib.tld"%>
<ias:cache>
<ias:criteria timeout="30">
<ias:check class="com.iplanet.server.servlet.test.Checker"/>
```

```
<ias:param name="y" value="*" scope="request"/>
</ias:criteria>
</ias:cache>
<%! int i=0; %>
<html>
<body>
<h2>Hello there</h2>
I should be cached.
No? <b><%= i++ %></b>
</body>
</html>
where Checker is defined as:
package com.iplanet.server.servlet.test;
import javax.servlet.*;
import javax.servlet.http.*;
public class Checker {
String chk = "42";
public Checker()
{
}
public Boolean check(ServletRequest _req)
HttpServletRequest req = (HttpServletRequest)_req;
String par = req.getParameter("x");
return new Boolean(par == null ? false : par.equals(chk));
}
}
```

Given the above, a cached element is valid for a request with parameter x=42, and y equal to the value used to store the element. Note that it is possible to have multiple sets of <ias:param> and <ias:check> inside an <ias:criteria> block. Also, its possible to have multiple <ias:criteria> blocks inside a JSP.

NOTE * cache-criteria = "*" does not work

* when cache-criteria is properly established (arg="*"), behavior is inefficient, i.e., the behavior is an update on cache miss, a cache hit, then an update, then a cache hit.

Tuning the Java Runtime System

You can tune the Java Runtime system by binding the application threads to Solaris® user level threads. The Solaris operating environment, by default, supports a two level thread model. Application level Java threads are mapped to user level Solaris threads, which are multiplexed on a limited pool of light weight processes (lwps). Often, we need only as many lwps as there are processors on the system, leading to conserved kernel resources and greater system efficiency. This helps when there are hundreds of user level threads.

In this chapter, we will discuss the following topic:

- Using Bound Threads
- Managing Memory and Allocation

Using Bound Threads

It is also possible to bind application threads to Solaris `lwps`, on a 1-1 basis. On some applications, better performance may be achieved by using the non-default model, when there are only a limited number of threads and hence few `lwps` being created. The JVM can be configured to map Java threads to bound Solaris threads before the KJS executable command is invoked in the KJS shell script:

```
_JVM_ARGS="bound_threads"

export _JVM_ARGS
```

Managing Memory and Allocation

The efficient running of any tool depends on how well memory and garbage collection is managed. The topics listed in this section will provide you with the information you need to optimize on memory and allocation functions.

In this section, we will discuss the following topics:

- Tuning the Garbage Collector
- Specifying Garbage Collector Setting
- Tracing Garbage Collection
- Tuning the Java Heap
- Tuning the Dynamic Compiler

Tuning the Garbage Collector

Newer Java Runtime Environments (JRE) come with a generational object memory system and sophisticated garbage collection algorithms.

A generational memory system, divides the heap into a few carefully sized partitions, called generations. The efficiency of a generational memory system is based on the observation that most objects are short lived. Newly allocated objects are allocated in the young generation (also referred to as the Eden). Because of the high mortality rates of newly allocated objects, scavenging or garbage collecting in the young generation is often very productive, quickly recycling a lot of allocation space.

Compacting garbage collectors use two semi-spaces in the eden, copying surviving objects from one young space to the second. Objects that survive multiple young space collections are tenured, that is, copied to a tenured generation. The tenured generation is larger and fills up less quickly. So it is garbage collected less frequently and each collection takes longer than a young space only collection. Collecting the tenured space is also referred to as doing a full GC.

The frequent young space collections are quick (few milliseconds) and the occasional full GC takes a relatively longer time (tens of milliseconds to even a few seconds, depending upon the heap size).

Other garbage collection algorithms such as the Train algorithm, are incremental, that is. they chop down the full GC into several incremental pieces. This provides a high probability of small garbage collection pauses even when full gc kicks in. This does come with an overhead and is usually not required for enterprise web applications.

Typically, a third generation, called the permanent generation is also created by the JVM to store internal objects such as loaded java classes.

Both HotSpot and Solaris JDK sport a generational garbage collection system. Only HotSpot ships with the incremental Train garbage collector. HotSpot is the default on both Solaris and NT platforms. In the future, new parallel and concurrent collectors will be introduced in JDK 1.4.

Both HotSpot and Solaris JDK use thread local object allocation pools for lock-free, fast, and scalable object allocation. User application level object pooling may have actually been beneficial when running on earlier generation Java Virtual Machines. But it might actually slow down the application, on this new generation virtual machines available with JDK 1.2 onwards. Consider pooling only if the object construction cost is very high and shows up being significant in the execution profiles.

Specifying Garbage Collector Setting

The following settings can be used to improve memory utilization by the Garbage Collector:

- Explicit Garbage Collector
- Deferred Garbage Collection

Explicit Garbage Collector

Memory utilization by the application server can now be controlled by modifying the behavior of the explicit Garbage Collector. You can enable or disable the cleaner by modifying the `JAVA_GX_ARGS=-DGX.cleaner.enabled` key in `iasenv.ksh` file on Solaris, and in the registry settings on Windows.

On Solaris, the `JAVA_GX_ARGS=-DGX.cleaner.enabled` entry is by default set to no, to improve response time for requests. If you notice an inordinately high usage of memory by iPlanet Application Server, enable the cleaner by changing the value to yes, or by commenting out the `JAVA_GX_ARGS` line.

```
JAVA_GX_ARGS=-DGX.cleaner.enabled=yes
```

Once enabled, the cleaner is invoked every 10 seconds. The cleaner interval can be controlled by adding the following line to `iasenv.ksh` file:

`JAVA_GX_ARGS=-DGX.cleaner.interval=N` where `N` denotes the time in milliseconds.

On Windows, the cleaner is enabled by default. To improve performance you can disable the cleaner or set a longer time interval.

If you want to modify or change the default behavior of the cleaner, it can be done by adding the following flags to the `JavaArgs` in the registry under, `SOFTWARE\iPlanet\Application Server\6.0\Java`.

```
-DGX.cleaner.doGC=yes -DGX.cleaner.interval=N
```

where `N` denotes the time in milliseconds.

Deferred Garbage Collection

A new switch has been introduced to enable deferred GC for Applogic based applications that see memory growth. This can be turned on by setting the java system property `useDeferredGC`.

On Solaris you can enable this by appending the following `JAVA_ARGS` in `iasenv.ksh` file:

```
-DuseDeferredGC=true
```

On Windows, append the following to the JavaArgs entry which can be found in the registry under `SOFTWARE\iPlanet\Application Server\6.0\Java` :

```
-DuseDeferredGC=true
```

The default value of this property is set as false, since high memory growth has been reported with a few Applogic based applications. In deferred garbage collection, the references of the newly created objects are stored temporarily till the end of execution of the request, to prevent its garbage collection. Therefore at the end of the request, all the memory is freed at once but it can also lead to transient (until end of request) memory growth.

Tracing Garbage Collection

Supplying `-verbose:gc` (sometimes `-verbosegc`) flag to the JVM will result in an informative one line message being printed out at every collection. It is useful to turn this flag on for a couple of reasons.

- It gives a log of all the garbage collection pauses and their length to detect if there are unacceptably long pauses.
- It serves as a "heartbeat" of sorts showing that the JVM is alive and well.

  Note that it is possible to get normal garbage collection even if the application logic is deadlocked.
- You can see if the application is leaking Java objects very easily. A memory leak is suspect if the number of non-garbage objects increases, even after many, full garbage collections.

Tuning the Java Heap

Now that we know what generational garbage collection is about, it is easy why heap configuration helps performance.

This section contains the following topics:

- Guidelines for Java Heap Sizing
- HotSpot Server VM Tuning Options
- Sample heap configuration on Solaris
- Sample Heap configuration on Windows

Guidelines for Java Heap Sizing

These are important guidelines for sizing Java heap.

- Determine how much of Java heap you can afford to give each JVM process.

  You can do this by first determining what is the amount of system memory that can be used by the application server node. Then divide it equally between the number of KJS processes that you wish to configure. Each KJS process is a JVM process.

  The rule of thumb for the ratio of KJS processes to the number of CPUs is one KJS per CPU. You could experiment with a little more or a little less.
- Set the starting and maximum Java heap size to the size that you have determined above. The JVM flags, `-Xms<size>` and `-Xmx<size>` flags specify the minimum and maximum heap size. Look at the JVM documentation for more details.

Example: *-Xms64m -Xmx64m* clamps heap size at 64m. Setting the starting heap size (`-Xms`) as well as the maximum allowed heap size (`-Xmx`) to the same value has a benefit. If the JVM were allowed to start with the default starting heap size, the heap size expands automatically. However, expansion is a slow process and during this heap expansion phase there would be frequent garbage collection and performance will be hampered.

Larger Eden or younger generation spaces increase the spacing between full garbage collections. But young space collections could take a proportionally longer time. In general, you could keep the eden size between 0.25 and 0.5 times the maximum heap size.

HotSpot Server VM Tuning Options

iPlanet Application Server 6.5 loads the 1.3 Hotspot Server VM by default. The Server mode VM is better suited for server side applications. iPlanet Application Server starts up with a tuned VM with the following arguments:

```
-server -Xss512k -Xms128m -Xmx1024m -XX:NewSize=42m
-XX:MaxNewSize=342m
```

```
-Xconcurrentio -XX:+DisableExplicitGC
```

You can increase or decrease the heap sizes based on the physical memory available.

These options may work well for some applications, whereas it might not for certain others. It depends on the type of application- whether its I/O bound, or compute intensive, or memory intensive, etc. You might have to experiment with the tunable parameters before deciding on the best options.

Table 5-1 gives the VM options and their descriptions.

Table 5-1 HotSpot Server JVM tuning options

VM Option	Description
-XX:NewSize=<n>	Initial new generation size (in bytes).
-XX:MaxNewSize=<n>	Maximum new generation size (in bytes).
-XX:+DisableExplicitGC	Disables explicit calls to GC, VM in total control of GC.
-Xconcurrentio	Uses LWP based synchronization instead of thread based synchronization.
-XX:CompileThreshold=<n>	'n' denotes the number of method calls after which further optimization is done by the hotspot compiler. The default value for server mode is 10,000 and 1,500 for client mode.

Table 5-1 HotSpot Server JVM tuning options

VM Option	Description
-Xbatch	Disables background compilation
-Xincgc	Incremental Garbage Collection.

Setting the young generation sizes to a constant value will prevent resizing. During our tests we found that using incremental GC reduces throughput.

For more information on JVM tuning, see http://java.sun.com/docs/hotspot/index.html.

Sample heap configuration on Solaris

Add the following arguments to the JAVA_ARGS environment variable in the KJS shell script:

```
-Xgenconfig:64m,64m,semispaces:64m,512m,markcompact
```

This creates a 512 MB Java heap, with two 64 MB semi-spaces for the young generation. We specify that a mark and compact algorithm be used. It is possible to increase the size of the young generation and the overall heap size. The Solaris JDK appears to allocate twice the amount as the specified size of each semi space, with the justification of accounting for object header and other space overheads.

`-Xgenconfig` is a complex flag to understand and get right. It is not well publicized and documented except in JVM internal articles.

`-Xgenconfig, -Xms, -Xmx` flags obviously interact and when specified together the `genconfig` setting overrides other settings. However, some range checking seems to be performed by the JVM to make sure that the minimum and maximum are consistent with what is specified to `genconfig`.

Sample Heap configuration on Windows

You can pass arguments to the JVM by setting the following property in iPlanet Registry:

`HKEY_LOCAL_MACHINE\SOFTWARE\iPlanet\Application Server\6.0\Java\JavaArgs`, to the desired string. This is where you set heap sizing parameters.

`-Xms` and `-Xmx` flags should be set as as described in the case of the Solaris JDK.

`-XX:NewSize=<size>` specifies the initial size, in bytes, of the young object space where new objects are allocated. The default initial young space size is 2MB. NewSize must be a multiple of `1024`. Append the letter k or K to indicate kilobytes, or m or M to indicate megabyte. `-XX:NewSize=64m` sets the initial size of the young space to 64mb. Note that a large young space size may result in increased garbage collection pause times.

`-XX:MaxNewSize=<size>` specifies the maximum size, in bytes, of the young object space where new objects are allocated. The initial young space size is 2MB. `MaxNewSize` must be a multiple of `1024`, and greater than `2MB`. Append the letter k or K to indicate kilobytes, or m or M to indicate megabytes. The default value for `MaxNewSize` is 64 MB. `-XX:MaxNewSize=128m` allows the young space to expand, if needed, to 128 MB.

`-XX:SurvivorRatio=k` sets ratio of eden size to survivor space size. For example the default ratio of 8 on Windows, with `NewSize=64m`, results in two semi spaces, each 4mb in size. Using `NewSize` and `SurvivorRatio` it is possible to get the desired semi-space size.

We recommend that you try this flag as a last resort, as it can disturb quite a few of the internal sizing calculations.

NOTE We have heard of about up to 30% performance improvements from a setting such as `-Xms256m -Xmx256m -XX:NewSize=128m -XX:MaxNewSize=128m`, compared to the default setting. Setting the `NewSize` and `MaxNewSize` appropriately can have a significant performance impact.

These `HotSpot` flags could apply all `HotSpot` based Java Runtime Environments. HotSpot based JDK 1.3 is supplied as default on Solaris, Windows and Linux. All HotSpot performance flags are listed at:
`http://java.sun.com/docs/hotspot/VMOptions.html`.

Tuning the Dynamic Compiler

Both Java HotSpot 1.3 for Windows and Solaris JDK 1.3.1 implement adaptive dynamic compilation, to detect program hotspots and compile only the hot program segments for peak performance. You are likely to observe a short ramp up, when this profile driven compilation takes place at application start up. For this reason be careful to make benchmark measurements on a warmed-up iPlanet Application Server.

Turning off the JIT is expensive. We have measured up to *three times* hit in performance on a well tuned system configuration when the JIT is turned off while using an application. Your mileage may vary, depending on the nature of the application and system bottlenecks on your hardware and database configuration.

In the following cases, you may wish to turn off the Dynamic Compiler:

- During debugging, to print and examine Java exception stack traces, annotated with source line numbers.
- To work around some rare Dynamic compiler bugs. In this case there are hidden JVM arguments with which you can selectively disable compiling a particular method or all methods in a class. Contact your Sun Java support for details.

If you do have to turn off the compiler, on Windows, supply `-Xint` in `JavaArgs` property of registry. On Solaris, add `-Djava.compiler=none` to `JAVA_ARGS` in the KJS shell script.

Chapter 6

Tuning the Operating System

There are a couple of Solaris® network performance tuning tricks that do not directly benefit iPlanet Application Server, but may be useful for other socket intensive customer applications that run in a data center application suite.

Tuning Solaris TCP/IP settings benefit programs that open and close a lot of sockets. iPlanet Application Server operates with a small fixed set of connections and the performance gain may not be as significant on the Application Server node. iPlanet Web Server and iPlanet Web Servers configured as a Web front-end to iPlanet Application Server can benefit significantly.

To retain these changes, after rebooting, set `ndd` variables in the file: `/etc/rc2.d/S69inet`.

This chapter discusses the following topics:

- Setting Time Wait Interval
- Setting TCP Connection Hash Table Size
- Binding Processes

Setting Time Wait Interval

After a connection has been closed by both the client and the server, the port remains unavailable for a certain amount of time, so that a new program does not inadvertently get packets that were intended for the old program. On Solaris machines, the default value of `tcp_time_wait_interval` is `240,000` ms (4 minutes). It is recommended that this be set at `60000` ms (1 minute) or even at `30000` (30 seconds) for better performance in socket communication intensive programs. The value can be modified and examined on a running system.

```
/usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 60000
```

```
/usr/sbin/ndd -get /dev/tcp tcp_time_wait_interval
```

Setting TCP Connection Hash Table Size

The connection hash table keeps all the information for active TCP connections (`ndd -get /dev/tcp tcp_conn_hash`). This value does not limit the number of connections, but it can cause connection hashing to take longer. To make lookups more efficient, set the value to half of the number of concurrent TCP connections that you expect on the server (`netstat -nP tcp|wc -l`, gives you a number). It defaults to `512`. This can only be set in `/etc/system` and becomes effective at boot time.

```
set tcp:tcp_conn_hash_size=8192
```

Binding Processes

Binding the Application Server processes to one or more processors used to produce significant performance gains (`20%-30%`) on earlier versions of iPlanet Application Server running on Solaris. However with SP2 and later release on Solaris 8, no significant gains were observed. We therefore do not recommend this setting.

Chapter 7

Tuning Database Servers

In this chapter section, we will discuss how to tune Oracle servers for maximum performance. We will also discuss how to tune Solaris to work with Oracle.

Tuning Oracle Servers

Oracle tuning is by itself a vast topic, but setting the following parameters correctly should be sufficient. Please refer to Oracle documentation for detailed information on each of the tunable configurations. This document is specifically for configuring oracle initial parameters on the Solaris® platform. These tips have been tested on Oracle 8.1.6 and later.

The first step is to set certain system shared memory pool parameters in `/etc/system` file. The Oracle architecture makes extensive use of shared memory segments for sharing data among multiple processes and semaphores for handling locking. The default kernel values may not be sufficient in most of the cases. The machine needs to be restarted after modifying `/etc/system` file. Typically these values should be sufficient, but may require some tweaking based on your system and resources:

You could use the command, `dbassist`, to create and tune the database. However, if the database instance is already created, then you will need to tune the parameters manually.

An Oracle server can be run in 2 modes - dedicated and shared server mode. Shared server or multi threaded mode enables many client user processes to share a small number of server processes.

All oracle initial parameters or tunable parameters are in this file:

`$ORACLE_HOME/dbs/init<SID>.ora`.

This file is actually a link to `$ORACLE_HOME/admin/<SID>/pfile/init<SID>.ora`

The values the installer would have created for the database instance may not be sufficient most of the time.

CAUTION Take a back up of this file before editing the entries.

Tuning Solaris Kernel Parameters

The following paragraphs describe how to tune Solaris Kernel parameters for Oracle:

- Make sure that the Solaris kernel has parameters set sufficiently high for Oracle. The Oracle architecture makes extensive use of shared memory segments for sharing data among multiple processes and semaphores for handling locking. Many operating systems, including Solaris, do not by default offer sufficient shared memory or semaphores for maintaining an Oracle database. However, you can change kernel parameters in Solaris simply by editing the `/etc/system` file and restarting the server.

Table 7-1 Solaris Kernel Parameters for Oracle

Kernel Parameter	Initial Setting	Purpose
SHMMAX	4294967295	Maximum size of a single shared memory segment.
SHMMIN	1	Minimum size of a single shared memory segment.
SHMMNI	100	Maximum number of shared memory segments in entire system.
SHMSEG	10	Maximum number of shared memory segments one process can attach.
SEMMNS	2000	Maximum number of semaphores in entire system.
SEMMSL	1000	Maximum number of semaphores per set.
SEMMNI	100	Maximum number of semaphore sets in entire system.
SEMOPM	100	Maximum number of operations per semop call.
SEMVMX	32767	Maximum value of a semaphore.

The first four kernel parameters configure shared memory segments. The recommended settings shown here should be appropriate for almost any Oracle database implementation. The SHMMAX setting may seem excessive, but there is no penalty to be paid by setting SHMMAX larger than you actually need.

The last five kernel parameters configure semaphores. Each Oracle instance requires one semaphore for each process, plus ten extras. Additionally, the largest instance requires a second semaphore for each process. If you will only be setting up one database on your server, the upshot is that you will need two semaphores for each process plus ten extras.

The recommended settings for the first two semaphore kernel parameters, SEMMNS and SEMMSL, should be appropriate for most Oracle implementations. For systems with large numbers of concurrent database connections, you may need to increase these values. The recommended settings shown here for the last three semaphore kernel parameters should be appropriate for just about any Oracle database implementation.

In general, if your Solaris kernel already has any of these parameters set larger than recommended here, you should not reduce the settings. If you do change any kernel parameter settings in /etc/system, then reboot the server so that the new settings will take effect.

Add the following lines to the end of your /etc/system file:

```
set shmsys:shminfo_shmmax=4294967295
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=100
set shmsys:shminfo_shmseg=10
set semsys:seminfo_semmns=2000
set semsys:seminfo_semmsl=1000
set semsys:seminfo_semmni=100
set semsys:seminfo_semopm=100
set semsys:seminfo_semvmx=32767
```

Chapter 8

General Guidelines for Better Performance

The following general guidelines are designed to elicit better performance:

- Change the multicast server host address and port number in a network to reduce the updates received and avoid writes to the registry where there are multiple iPlanet™ Application Server installations in a network which are not in a single cluster.
- If you are using the native JDBC driver for Oracle, then to enable parsing of SQL statements during `OPARSE` call, set the environment variable `IAS_OPARSE_NODEFER` in `iasenv.ksh` file. If this is not set, parsing will be deferred until `oexec`.

  iPlanet Application Server with Oracle 8.1.6 core dumps if an illegal SQL statement is passed. Setting this variable can avoid this occurrence. This setting is only recommended for development environment and not for production environment. This feature is available from iPlanet Application Server, Enterprise Edition 6.0, SP3 onwards.

Guidelines For Better EJB Performance

The following guidelines are designed to promote better EJB performance:

- If you don't need session failover, turn DSync off for your application. In other words, use lite rather than distributed session if you can, so that DSync doesn't have to propagate session changes to other machines.

- If you need session failover, but don't need protection for an entire machine failure, make sure you have multiple KJS process and use `Dsync-Local` sessions. Session failover will then be available for all KJS processes inside the iPlanet Application Server instance. However, session failover across machine boundaries will not be provided. At least your session information will survive a KJS crash.
- Make your session small -- about 4KB per user session is a heuristic that has been around for a long time. However, with iPlanet Application Server it appears that DSynch processing, not the size of the synch chunks, may be the performance governor. Therefore, it may be more important to keep the clusters small (4 instances or less), than it is to keep the user session-sizes small.
- Keep what you store in a session simple if distributed sessions are used. The iPlanet Application Server session service was designed for fairly simple name-value pairs.
- Serialization and deserialization is required to move session info around in a distributed session environment, so don't create big, complex objects that are expensive to marshall around.
- Don't rely on `HttpSession` time-outs to clean up sessions. Use the `HttpSession.invalidate()` method to manually cleanup where possible.
- Set the `HttpSession` time-out as low as possible so that the container gets a chance to clean up an unwanted session sooner.
- If handles to stateful session EJBs are being stored in `HttpSessions` by servelets or JSP's accessing them, then the EJB session timeout should be set close to the `HtttpSession` timeout.

  The default session time-out is `14400` seconds (4 hours) for stateful session EJBs. Set this to a more reasonable value. This will prevent KJS memory growth and avoid unnecessary passivation of EJB instances that are no longer needed.
- Specifically for JSPs, if your JSPs are not explicitly using HttpSession, you are creating unnecessary `HttpSession` objects since a session object is created for you automatically. To prevent this, add the following JSP page directive to your JSPs: `<%@session=false %>`.

Chapter 9

Validating Server Performance

You can monitor and validate server performance regularly to ensure that the methods that you've set in place for high performance are being carried out. The following topics will help you validate iPlanet™ Application Server performance.

In this chapter, we will discuss the following topics:

- Monitoring iPlanet Application Server
- Using Performance Tuning Tools
- Setting Up SNMP Monitoring
- Obtaining Performance Data

Monitoring iPlanet Application Server

Monitoring comes under the serviceability part of RAS (Reliability, Availability and Serviceability). Service provisioning is not possible unless we know the dynamic service behavior.

Service providers want to have real time monitoring of application performance, as experienced by users, accessing externally exposed web services. A data center is likely to have monitoring consoles with graphical displays and alert systems that help maintain Service Level Agreements.

Therefore, application and web service containers must provide accurate and easily accessible performance information to such tools. Such monitoring can be at various levels, but at the minimum, give a view of the rate at which requests are being processed in the server complex.

Monitoring should be flexible, that is, it should be possible to turn it on and off in a running server, without restarts. The data must be well formatted so it can be coupled with various other higher level monitoring tools.

Basically, a well-tuned system should show the following performance:

- An even usage of CPU time across all of the servers.
- An even usage of CPU time across all processors in each server.
- An even usage of CPU time across all of the KJS or KCS processes.
- A fairly low percentage of system time (0 - 25%), especially if the workflow is computationally intensive.
- Full utilization of all processors allocated to the KXS process.

Use the iPlanet Application Server Administration Tool (iASAT) to monitor KXS and KJS/KCS statistics. Look for requests waiting while the processors are not completely busy to indicate that the server is not tuned optimally. Check active database connections in the KJS/KCS engines to help set the appropriate number of connections for the DB connection pooling.

On Solaris

For a standard installation, it is difficult to observe the amount of processing time used by each process. However, if you `pbind` each of the iPlanet Application Server processes to a separate CPU, `mpstat` will show the percentage of time each CPU spends. `mpstat` will also display the ratio of user time to system time for each of the processors. `proctool` (available at `http://www.sunfreeware.com`) is a very handy tool that shows resource usage per process. Additionally, it provides a GUI for binding processes to processors and for changing process priorities.

This section includes the following topic:

- Adding Plots Using iASAT

Adding Plots Using iASAT

To monitor process attributes and validate process performance, you can add plots using iASAT. These plots help you to chart KJS, KCS and KXS process attributes. For information on how to configure these plots, and lot process attribute data, see Chapter 2, “Monitoring Server Activity”, in *iPlanet Application Server Administration Guide*.

Using Performance Tuning Tools

Many commercially available tools can be used to profile the behavior of J2EE applications running on iPlanet Application Server. Most of these tools rely on the JVM Profiling Interface (JVMPI) to obtain dynamic information from a running Java Virtual Machine.

Machine Process, JProbe and OptimizeIt are examples of this class of tools. These tools are used to profile CPU and Memory utilization, inspect objects, detect application memory leaks, detect deadlocks, perform code coverage and other troubleshooting activities, in a development environment. JVMPI adds significant performance overhead and is not suitable for monitoring and profiling deployed Applications.

Selective bytecode instrumentation can be used for more targeted and efficient performance monitoring. While not as informative or powerful, instrumentation can provide more realistic data. Introscope from Wily Solution is an example of such technology, integrated into iPlanet Application Server, from Version 6.0, SP3 onwards.

This section lists procedures for using JProbe, OptimizeIt and Introscope with iPlanet Application Server. We will stick to the Windows version of Application Server, but these products do exist and work in a similar fashion on Solaris.

This section includes the following topics:

- Tuning Performance Using OptimizeIt
- Tuning Performance Using JProbe
- Tuning Performance Using IntroScope

Tuning Performance Using OptimizeIt

OptimizeIt is a product that you can use to tune iPlanet Application Server performance. You can download the product from *http://www.optimizeit.com*. To use this tool, perform the following tasks:

- Configure the JVM to use the JDK bundled with iPlanet Application Server (JDK 1.3.1_02 with HotSpot).
- Add `com.kivasoft.engine.Engine` as the name of the class file to be invoked.
- Add all the entries separated by a semicolon (`;`) that you find in the classpath entries in iPlanet Registry, to the classpath in OptimizeIt. You are now ready to start profiling.

On Solaris, the steps are very similar to those on Windows NT. However, OptimizeIt is certified to work only with the Reference JDK 1.2 for Solaris and this is not bundled with iPlanet Application Server 6.0.

Tuning Performance Using JProbe

JProbe is a third party tool that you can use to tune iPlanet Application Server performance. To know more about how to install and use Jprobe with iPlanet Application Server, go to `http://www.jprobe.com/software/support/jprobe/j2ee/iplanet.html`.

Tuning Performance Using IntroScope

Introscope needs to instrument all the desired application class files to gather profile information. This instrumentation can be done statically by hand or dynamically, by integrating this with iPlanet Application Server internal class loaders.

We recommend the second approach because it is more seamless and can be dynamically controlled without changing the deployed bytecodes on disk. To learn more about Wily Tech's `Introscope, visit http://www.wilytech.com`.

Integrated support for Introscope, with dynamic bytecode instrumentation, is available in iPlanet Application Server, Enterprise Edition 6.0, SP3 and later.

Setting Up SNMP Monitoring

SNMP is a protocol used to exchange data about network activity. With SNMP, data travels between your application server and a workstation where network management software is installed. From this workstation, you can remotely monitor your network and exchange information about network activity between servers. For example, using an application like HP OpenView, you can monitor which iPlanet Application Server machines are running, as well as the number and type of error messages your application servers receive.

Your network management workstation exchanges information with the application servers in your enterprise through two types of agents: the subagent and the master agent. The subagent gathers information about an application server and passes that information to the master agent. The master agent exchanges information between the various subagents and the network management workstation. The master agent runs on the same host machine as the subagents with which it communicates.

To know more about how to set up SNMP monitoring, see “Configuring SNMP to Monitor iPlanet Application Server Using Third-Party Tools”, in *iPlanet Application Server Administrator’s Guide.*

Obtaining Performance Data

The web request process flow passes from the load generator to the Web server front-end, and through the in-process iPlanet Application Server Plugin to the KXS process and onwards to a target KJS process for execution. The response retraces a similar path. The trip time can be computed at 4 points.

At the Load Generator. If you are using tools like SilkRunner or another custom load generation tool, these tools already provide measurement and graphing methods. These tools measure the actual response time as would be experienced by real users.

At Web Server. It is possible to configure the iPlanet Web Server to gather response time profiles. Consult the Web Server Performance tuning and Analysis Guide.

At iPlanet Application Server Web Plugin. This measures the amount of time iPlanet Application Server takes to respond to a request. It is measured from the perspective of the web connector - the time a request is sent out to iPlanet Application Server till the time a response is received at the Web-Connector Plugin, executing in the web server.

To get these logs, enable the following key at the iPlanet Web Server server in iPlanet Registry:

```
SOFTWARE\iPlanet\Application Server\6.0\CCS0\HTTPAPI\NASRespTime=1.
```

This will dump timing statistics on the web server’s log files. To extract timing information, execute a composite shell command that could look similar to:

```
grep "plugin reports" errors| grep -v Registry| cut -c2-21,64- | cut
-d " " -f1,2,4
```

A sample output is shown below. The time is measured in milliseconds:

```
22/Sep/2000:19:36:09 </NASApp/tmf/TMFServlet> 420

22/Sep/2000:19:36:10 </NASApp/tmf/TMFServlet> 600

22/Sep/2000:19:36:16 </NASApp/tmf/TMFServlet> 392

22/Sep/2000:19:36:16 </NASApp/tmf/TMFServlet> 220

22/Sep/2000:19:36:16 </NASApp/tmf/TMFServlet> 428
```

The KXS logs at the iPlanet Application Server server (located at `<IAS_HOME>/ias/logs/KXS`) maintain the `reqstart` and `reqexit` times of each request. The `reqexit` value provides the processing time of each request. This can be used to measure the time taken at iPlanet Application Server to execute a servlet or JSP request.

Setting the following key to 1, in iPlanet Registry, is a useful monitoring trick:

```
Software\iPlanet\Application Server\6.0\CCS0\REQ\debug=1
```

You should start to see entries in the KXS log, for example:

```
[26/Apr/2001 11:48:05:7] info: NSAPICLI-012: plugin reqstart,
tickct:

988310885s 763786us

[04/26/01 11:48:05:768] Request 00 Starting AppLogic

{1A488137-7510-1941-BAE5-080020B90F48} on Engine 0

[04/26/01 11:51:07:504] Request 00 Completing AppLogic

{1A488137-7510-1941-BAE5-080020B90F48} Execution

[26/Apr/2001 11:51:07:5] info: NSAPICLI-009: plugin reqexit: 181s

741781us

Request # starts at 00 and increments
```

Using data gathered at these probe points, it is possible to accurately determine the time taken for each leg of the request round trip. If the number of threads processing requests in any queue along the path is inadequately configured, queuing delays will dominate the response time. Configure iPlanet Application Server and iPlanet Web Server as we have suggested in this document, and focus attention on application performance tuning.

Chapter 10

Frequently Asked Questions

Over the years, iPlanet™ system engineers, professional services consultants and customers have experimented with procedures to optimize the iPlanet Application Server production environment. The resulting set of thumb rules have been compiled and reproduced in this chapter.

Frequently Asked Questions (FAQ) related to peformance have been categorized in the following sections:

- Environment Setup
- System Tuning
- Application Tuning

We welcome your feedback as we continue to refine and expand this FAQ.

Environment Setup

This section deals with issues related to setting up the iPlanet Application Server environment.

1. How much RAM do I need per iPlanet Application Server CPU, in a typical production environment?

You will need 1GB of RAM per installed CPU of iPlanet Application Server.

2. How many CPUs are needed for a team of developers, in a typical developer sandbox installation on Solaris.

A single iPlanet Application Server CPU can support 3-5 developers. If usage is intense, about 2-3 developers per CPU.

3. How much disk space do I need to install iPlanet Application Server?

   The distribution is about 150 MB and you should count on three times that space for installation and operation. You will need about 450 MB of free disk space initially. After installation, iPlanet Application Server will need about 256 MB of disk for operation.

4. How many processors can a single iPlanet Application Server instance be expected to utilize efficiently?

   As a general rule, each instance should run no more than 8 to 12 processors. A single iPlanet Application Server instance has only one KXS, but can have many KJS processes. If the application seems to be constrained by KJS rather than KXS, you can scale into the 12 processor range. However, if you make heavy use of a KXS-based service like distributed sesson management (DSync) and servlet result set caching, KXS will limit scalability to the 8 processor range. To scale beyond this 8 to 12-processor range, consider installing multiple iPlanet Application Server instances on the machine.

5. iPlanet Application Server distributed session management (DSync) facility can have a high overhead. What key decision can I make at installation time to help lessen the load, distributed session management (DSync) has on the KXS process?

   Elect to have only one DSync backup. This will reduce the amount of DSync work required to keep the backup's in-memory session store synchronized with the master so that it is available to take over in the event of the primary failing. One DSync backup should be sufficient to assure ongoing session availability.

6. What can a Java servlet/JSP programmer do to help iPlanet Application Server's session management facility to be as efficient as possible?

   Look at your use of sessions. The DSync facility, housed in the KXS process, can put quite a load on your environment. Here are some considerations:

   - Limit your cluster size to no more than 4 instances. Beyond this size, the overhead of keeping the distributed session store in synch typically becomes a performance limiter.
   - Use sticky load balancing so subsequent processing for a user during a session always returns to the same KJS where session information is available locally. This is particularly important for a stateful session.

7. How can an administrator help KXS handle session management housekeeping chores more efficiently?

   As sessions timeout, or are invalidated, they need to be promptly removed from the in-memory session store. This removal can be done with greater concurrency by increasing the number of threads dedicated to session management. Adjust the following property in iPlanet Registry to make session node management more efficient:

```
SOFTWARE/iPlanet/Application
Server/6.0/Clusters/<machine-name>-NoDsync/SyncTimeoutThreadCount
```

System Tuning

1. What constitutes a well-tuned iPlanet Application Server system?

   A well-tuned system should show:

   - Uniform CPU time across all servers.
   - Even usage of CPU time across all processors in each server.
   - Comparable CPU time across all KJS processes.
   - A system time of (0-25%) if workflow is computationally intensive.
   - Utilization of all processors allocated to the KXS process.

2. There is a temptation to install iPlanet Application Server a number of times on a multiple-CPU machine. Is that the best way to scale?

   Install multiple iPlanet Application Server instances on a single machine as a last resort. This usually complicates the ongoing management. Instead, try tuning a single iPlanet Application Server instance first by adding additional KJS (Java VM) processes and tweaking the number of threads available in iPlanet Application Server process thread pools.

3. How many threads should a KXS process have?

   32 threads per KXS process is the default setting. However, you can increase the number of threads. Increasing the numbering of threads for KXS, however, does not ensure a dramatic improvement in performance.

   A well-tuned system should show:

   - Uniform CPU time across all servers
   - Even usage of CPU time across all processors in each server

- Comparable CPU time across all KJS processes
- A system time of 0-25%) if workflow is computationally intensive

It is important that all processors allocated to the KXS proces pool are utilized. When compared with KJS threads, KXS threads don't perform a lot of work, so you don't need as many KXS threads as KJS threads. To increase KXS performance, you can consider binding (`pbind`) KXS to a process or a processor set.

4. When should you consider binding KXS to a processor?

   There are benefits especially in a Solaris environment in binding iPlanet Application Server processes to specific processors or to a processor set. On a multi-processor machine, always bind KXS to a processor. This will significantly improve throughput due to overhead associated with using mutex locks on multiprocessors.

5. When should you consider binding KXS to a processor set (multiple processes)?

   If you still see that KXS is queuing requests, create a a processor set containing two processors. Bind the KXS to this processor set. Note, going beyond two processors in a process set for KXS typically doesn't lead to substantial improvements in throughput.

6. What should make you want to drop a processor from a process set bound to KXS?

   If KXS is not fulling utilizing the processor set and you see that there are KJS processes that are starved for power (threads are queuing), take away a processor from KXS and make it available for a new KJS process that you can add to your iPlanet Application Server configuration.

7. How many threads should a KJS have?

   Try starting at 32 and increasing from there as processor load gets high. 48 seems to be about as large a thread pool as you will want to set for KJS. Going much beyond that will simply increase context switching resulting in wasted cycles.

8. When should you consider binding KJS to a processor?

   Generally, you shouldn't bind KJS to processors or processor sets since doing so has shown to provide no more than a 5% performance increase. In JDK 1.2.2 and and beyond, the VM is already optimized to work well in multi-processor environments.

9. The KJS process is a "home" for the Java VM. How can I tailor things like heap size for the JVM that KJS hosts?

Arguments that can be supplied to the Solaris JDK may be set via the `JAVA_ARGS` shell variable in the `iasenv.ksh` shell script. Settings with JVM flags, particularly -Xms and -Xmx flags, specify the starting and maximum heap size used by each KJS engine. Heap size decisions should be based on how much memory is available on the system. Set these as large as possible without starving other applications running on the same server. The default starting heap size is 8MB.

The heap grows automatically as needed. Starting with a large heap size avoids frequent garbage collection during growth. To cap the heap growth size, use the `-Xmx` flag. The JDK documentation provides more information on these and other flags.

10. Your application seems to be behaving badly because it takes a long time between hitting the submit button and seeing the result in my browser. How can you troubleshoot where the slowdowns may be occurring?

Think about using "clocks" at key points in the request processing cycle:

- Clock 1 timestamps at the client (browser or load generation tool like `LoadRunner`).
- Clock 2 timestamps on the iWS web server at the point where a front-end thread receives a request.
- Clock 3 timestamps on the iWS web server at the point where the back-end worker thread gets the request.
- Clock 4 timestamps when the iPlanet Application Server web connector actually responds to the request from the back-end worker thread.
- Clock 5 timestamps when a request is received by KXS to be sent to a KJS for processing.
- Clock 6 timestamps when KXS receives the request back from KJS.

11. If there is a long delay between Clock 1 and Clock 2, what might you suspect?

The delay could be because of one of the following reasons:

- Network congestion.
- CPU or NIC too busy on the client side.
- TCP stack queuing is occurring at the web server.

- Requests may be queuing at an intermediary, such as a load balancer or a firewall.
- Misconfiguration of your load generation client.

12. What if you see a delay of greater than two seconds between Clock 2 and Clock 3?

It probably means that all the back-end worker threads are busy and requests are being queued. You can either increase the number of back-end threads or decrease the number of front-end threads.

13. If there is a delay of greater than 30 milliseconds between Clock 3 and Clock 4, what will you have learned?

Well, not much from a corrective action point of view. This indicates that the web connector isn't performing as fast as anticipated. Unfortunately, there aren't any tuning opportunities here.

14. What if you see elapsed times of greater thand 3 seconds between Clock 4 and 5?

Suspect network buffering at the web server NIC or queuing at the iPlanet Application Server NIC. You might also check for an under-resourced firewall. Also, general network congestion can be the problem.

15. What if the Clock 5 and Clock 6 timestamps reveal slower turnaround than anticipated?

It tells you that the KXS and / or KJS processes need some tuning. Check the thread pools for KXS and KJS. Also, consider adding additional KJS processes. Beyond that, you need to consider binding iPlanet Application Server processes to processors and maybe even to processor sets.

Application Tuning

1. Is their some tweaking you can do as an EJB programmer to help performance of iPlanet Application Server EJB container?

- The default passivation timeout is 60 seconds. Increase this value if the bean instance creation rate is very low and the bean size is large. You may see a slight boost in performance as passivation processing is reduced.
- Make Meta Data Cache Size as large as the number of different types of beans that exist in your application. The default, 30, may not be enough to cache all the home handles.

- Set `Implementation Cache Size` based upon the number of concurrent user sessions you expect. For example, if each 200 concurrent user sessions will have a need for one stateful session EJB, set the `Implementation Cache Size` to at least 200.

2. What can I check in my Java code that might help me make iPlanet Application Server perform better?

iPlanet Application Server KJS processes provide the J2EE containers inside which your Java code runs. The KJS processes host independent Java VMs. Therefore, you need to follow good Java guidelines to help the VMs perform as well as possible. Here are some things to check:

- Avoid serializing / deserializing. These are expensive operations.
- Avoid using lots of arrays since there is overhead to do what Java does for arrays, namely - initializing them for you and preventing you from accessing out of range.
- Explicitly dereference variables by setting them to null to make garbage collection more efficient
- Don't use class variables (static members) in servlet classes since they impose synchronization in the server. By default, all users share a single copy of the servlet code per web container (JVM).
- Avoid using sychronized methods or synchronized blocks in your code.
- Carefully consider EJB use. EJBs put a load on a server.

3. For performance reasons, you should use EJBs judiciously in your application; so, what's wrong with EJBs?

EJBs are wonderful. After all, they are *the* component model for server-side Java. However, all good things come with a price. This is true in any vendor's EJB server. There is quite a lot of overhead required to provide access to EJBs and to host the services that EJB containers provide, such as, security and transaction management. Here are some thoughts on minimizing performance degradation when using EJBs:

- Cache EJB references at the servlet so you won't need to do a JNDI lookup for every request.
- Use sticky load balancing so subsequent requests during a session always return to the same KJS (VM) for processing. The EJB resource can be local there and expensive calls across machines to get EJB access can be avoided.
- Use stateless session EJBs as they are comparable to servlets in performance.

- Stateful session and entity beans are much more expensive than stateless session EJB beans. Entity EJBs are most performance intensive with BMP being more expensive than CMP for persistence management of entity EJBs.

4. How much can servlet HTML result caching help the performance of your application?

   Servlet caching can almost double performance for a 1-CPU installation, but tests have shown only 4% enhancement on a 2-CPU iPlanet Application Server instance. At 4 CPUs and beyond, the cache doesnt scale as well and we don't recommend that you use servlet result caching. Use JSP result caching instead.

   JSP result caching is KJS-based rather than KXS-based. You can have multiple KJS processes per iPlanet Application Server instance so this approach scales better.

5. How can you make servlet HTML result caching most effective?

   You can enhance the possibility for cache hits by increasing the amount of memory available for storing the cache. This is done in the deployment descriptor for servlet caching and in the Administration Tool for JSPs.

   HTML result caching is handled by KXS for servlets, but that's one more thing to ask a probably already overworked KXS to do. HTML result caching can really enhance throughput.

6. JSP result caching.

   This cache is handled by KJS. Unlike KXS processes, you can have multiple KJS process per iPlanet Application Server instance. So, you can offload KXS and bring to bear more result caching power by going the JSP result caching route.

Index

F

G

H

I

J

K

L

M

U

V

W