WebSphere Application Server for iSeries

Performance

Version 5.0.2

eserver

WebSphere Application Server for iSeries

Performance

Version 5.0.2

Note

Before using this information and the product it supports, be sure to read the information in "Notices," on page 79.

Fifth Edition (September 2004)

This edition applies to version 5.0.2 of IBM WebSphere Application Server for iSeries (product number 5733-WS5) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 1998, 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Performance 1
Performance overview 2
Performance guidelines 2
Performance tools 3
Performance resources 5
Performance Monitoring Infrastructure (PMI) . . . 8
  Performance data organization 9
  Performance data classification 9
  PMI data counters 11
Monitor performance with Tivoli Performance Viewer 13
  Start Tivoli Performance Viewer 14
  Set performance monitoring levels 16
  View performance data 16
  Save performance data to a log file 18
  Enable Java Virtual Machine Profiler Interface (JVMPI) data reporting 19
Enable performance monitoring services 20
  Enable performance monitoring services for application servers 20
  Enable performance monitoring services for node agents 20
  Enable performance monitoring services with wsadmin 21
  Enable performance data collection with the administrative console 22
Measure data requests with PMI Request Metrics . . 22
  Enable PMI Request Metrics 23
  Enable Application Response Measurement . . . 23
  Configure Request Metrics trace filters 24
  Set the trace level for Request Metrics 25
  Request Metrics data output 25
Develop performance monitoring applications . . . 26
  Develop performance monitoring applications with the PMI client 27
    The PMI client package 27
    The PMI client interface 28
    Example: PMI client with new data structure 29
  Develop performance monitoring applications with the PMI servlet 35
    PMI servlet 36
  Access PMI data with the JMX interface 38
  Run monitoring applications. 40
  Run monitoring applications with security enabled. 41
Primary tuning parameters 42
Tuning parameters for your WebSphere Application Server environment. 42
  Application server tuning parameters. 43
  Java Messaging Service tuning parameters . . . 50
  Queuing network 54
    Queue configuration tips 56
    Enterprise bean method invocation queuing 58
  Web server plug-in tuning tips 59
  Web services tuning tips 60
  Hardware capacity and configuration. 62
  Java virtual machine tuning parameters 63
    Java memory tuning tips 65
  Web server tuning parameters 70
  Database tuning parameters 71
  TCP/IP buffer sizes. 71
Application assembly performance checklist . . . 72
Troubleshoot performance 73
Performance advisors 74
  Use the Performance Advisor in Tivoli Performance Viewer 75
  Use the Runtime Performance Advisor 76

Appendix. Notices 79
Trademarks 80
Terms and conditions for downloading and printing publications 81
Code example disclaimer 81

© Copyright IBM Corp. 1998, 2004

Performance

WebSphere Application Server provides the Performance Monitoring Infrastructure (PMI) to collect data for the application server run time and enterprise applications. You can use a variety of tools to monitor and analyze the performance data. Based on the data, you can tune various parameters to improve performance. WebSphere Application Server also includes performance advisors that provide additional tips to help you optimize performance.

- Introduction to WebSphere Application Server performance (page 1)
- Monitor performance (page 1)
- Tune performance (page 2)

Introduction to WebSphere Application Server performance

These topics provide general information and resources for performance management:

"Performance overview" on page 2

This topic provides an overview of performance concepts and guidelines.

"Performance guidelines" on page 2

This topic provides general guidelines that can help you ensure optimal performance.

"Performance tools" on page 3

This topic provides links to several tools that you can use to monitor and tune performance.

"Performance resources" on page 5

This topic provides links to additional resources that can help you optimize performance.

Monitor performance

These topics describe how to use the Performance Monitoring Infrastructure (PMI) to monitor performance, and provide information about the tools that you can use to monitor performance:

"Performance Monitoring Infrastructure (PMI)" on page 8

This topic describes PMI and provides information about data organization, data classification, and PMI data counters.

"Monitor performance with Tivoli Performance Viewer" on page 13

This topic describes how to monitor performance data with Tivoli Performance Viewer (formerly Resource Analyzer).

"Enable performance monitoring services" on page 20

This topic describes how to enable performance monitoring services for application servers and node agents, and how to enable performance data collection.

"Measure data requests with PMI Request Metrics" on page 22

Performance Monitoring Infrastructure (PMI) Request Metrics collect data by timing requests as they travel through WebSphere Application Server components. This topic provides information about how to use PMI Request Metrics to measure performance.

"Develop performance monitoring applications" on page 26

This topic describes how to use the PMI client package and the PMI servlet to develop performance monitoring applications.

© Copyright IBM Corp. 1998, 2004

Tune performance

These topics describe parameters that you can adjust to tune application performance, and describe the performance advisors that WebSphere Application Server provides:

"Primary tuning parameters" on page 42
This topic lists the primary parameters that can affect performance.

"Tuning parameters for your WebSphere Application Server environment" on page 42
This topic provides detailed information about parameters that can affect performance and how to change them.

"Application assembly performance checklist" on page 72
This topic describes settings that you can configure during application assembly to optimize application performance.

"Troubleshoot performance" on page 73
This topic describes some common performance problems and the actions that you can take to correct the problems.

"Performance advisors" on page 74
This topic describes the Runtime Performance Advisor and the Performance Advisor in Tivoli Performance Viewer.

Performance overview

Performance is the measure of time and resources required to complete a task. The goal of performance tuning is to decrease the amount of time and resources so that your application server can complete more tasks in less time. Because of the variety of resources involved, performance tuning for Web-based applications is more complex than tuning most other applications.

To understand performance management, you need to know these basic performance concepts:
- **Throughput** is the number of client requests that the application server environment can process at a given time.
- **Response time** is the elapsed time that it takes for the application server to process a client request.
- **Load** is the amount of resources, such as main storage, processor, and I/O support, that are required by the application to process all client requests at a given time.

Application server performance is determined primarily by these components:
- **Hardware resources** such as system configuration, memory pools, and subsystems
- **Web environment configuration** such as garbage collection, request queuing, and data caching
- **Application design and implementation** such as object creation, connection pooling, and data access configuration

"Performance guidelines" provides general guidelines that can help you ensure optimal performance.

Performance guidelines

This page provides basic guidelines that can help you ensure optimal performance. If you experience problems with performance, see Tune performance.
- Verify that you have enough system capacity. For more information, see these topics in *Installation*:
  - iSeries prerequisites for installing and running WebSphere Application Server
  - Prerequisites for installing and running WebSphere Application Server Network Deployment

You can also use the IBM eServer Workload Estimator or obtain professional services.

- Verify that hardware resources are allocated efficiently.
  - Provide sufficient memory and activity level in the *BASE memory pool.
  - If availability and performance are high priorities, it is recommended that you allocate WebSphere Application Server processing to a separate storage pool.

  You can adjust memory pools manually or automatically.
- Verify that you have the latest WebSphere Application Server Group PTF. For information about PTFs and to download the most recent Group PTF, see WebSphere Application Server for iSeries: PTFs.
- Tune the JVM garbage collector. For information about garbage collection, see Performance resources (page 6).
- Configure queues for your application server components. See these topics for more information:
  - "Queuing network" on page 54
  - Thread pool settings
  - Connection pool settings
- Configure dynamic caching for your application server. For more information, see Dynamic caching in *Application Development*. IBM HTTP Server (powered by Apache) provides a local cache that can store static content such as images and static Web pages. For more information, see the Manage server performance for HTTP Server (powered by Apache) topic in *Web serving*.
- If your application server uses public key encryption, use hardware accelerators to improve performance. For information about the hardware accelerators that are available for iSeries, see the appropriate document based on your version of OS/400:
  - For V5R1: iSeries Hardware Cryptography Performance
  - For V5R2: iSeries Cryptographic Offload Performance Considerations
- Design applications to optimize performance. Performance resources (page 6) includes links to resources that can help you develop efficient Java applications.
- Pool network connections. For more information, see Connection pooling in *Application Development*.
- Use efficient SQL and JDBC to optimize data access.
- Monitor and analyze run-time behavior. WebSphere Application Server includes a stand-alone Java client called Tivoli Performance Viewer. You can also develop your own performance monitoring applications. For more information, see these topics:
  - "Monitor performance with Tivoli Performance Viewer" on page 13
  - "Develop performance monitoring applications" on page 26

Performance tools

Several tools are available that can help you monitor and tune WebSphere Application Server performance. These Web sites and Information Center topics provide information about the tools:

"Monitor performance with Tivoli Performance Viewer" on page 13
Tivoli Performance Viewer (formerly Resource Analyzer) is a stand-alone Java application that you can use to monitor WebSphere Application Server performance.

"Use the Performance Advisor in Tivoli Performance Viewer" on page 75
The Performance Advisor in Tivoli Performance Viewer provides information to help you optimize performance. This topic describes how to configure and use the Performance Advisor.

“Use the Runtime Performance Advisor” on page 76
The Runtime Performance Advisor provides information to help you optimize performance. This topic describes how to configure and use the Runtime Performance Advisor.

“Measure data requests with PMI Request Metrics” on page 22
Performance Monitoring Infrastructure (PMI) Request Metrics collect data by timing requests as they travel through WebSphere Application Server components. This topic provides information about how to use PMI Request Metrics to measure performance.

“Develop performance monitoring applications” on page 26
You can use the PMI client package and the PMI servlet to develop custom performance monitoring applications.

WebSphere Performance Management Business Partner Solution
This page provides links to IBM business partners who offer performance monitoring tools that you can use with WebSphere Application Server.

WebSphere Application Server Performance Tuning and Analysis Tools
This high-level online course is offered through IBM eServer Solutions Enablement. The course describes elements that can affect WebSphere Application Server performance, provides basic guidelines to help you acheive optimal performance, and discusses various available tools that you can use to analyze and tune performance.

The ANZJVM (Analyze Java Virtual Machine) command
The ANZJVM command collects information about the Java Virtual Machine (JVM) for a specified job. This command is available in OS/400 V5R2 and later.

The Dump Java Virtual Machine (DMPJVM) command
This command dumps JVM information for a specified job.

Collection Services
You can use Collection Services to collect performance data, which you can analyze with other performance tools. See this Information Center topic for more information.

Performance Tools for iSeries
Performance Tools for iSeries is a set of tools and commands that you can use to view and analyze iSeries performance data in several ways. See this Information Center topic for more information.

Performance Explorer (PEX) Information Center topic
Performance explorer is a data collection tool that helps you identify the causes of performance problems that cannot be identified with the other available tools or general trend analysis. See this Information Center topic for more information.

Performance Explorer (PEX) Web site
This Web site provides additional information about PEX.

Performance Data Collector tool
The Performance Data Collector (PDC) tool provides profile information about the programs that run on the iSeries server. See this Information Center topic for more information.

IBM Performance Management for eServer iSeries Information Center topic
IBM Performance Management for iSeries (formerly known as PM/400) uses Collection Services to gather the nonproprietary performance and capacity data from your server and then sends the data to IBM for analysis. See this Information Center topic for more information.

IBM Performance Management for eServer iSeries Web site
This Web site provides additional information about PM eServer iSeries.

Performance Trace Data Visualizer
Performance Trace Data Visualizer (PTDV) for iSeries is a tool for processing, analyzing, and viewing Performance Explorer collection data residing in PEX database files.

iDoctor for iSeries
iDoctor for iSeries is a suite of applications that can help you monitor performance and troubleshoot common problems on your iSeries server.

Heap Analysis Tools for Java(TM) This tool is a component of the iDoctor for iSeries suite of performance monitoring tools. The Heap Analysis Tools component performs Java application heap analysis and object create profiling (size and identification) over time. This tool is sometimes called Java Watcher or Heap Analyzer.

Tivoli: Performance and Availability Solutions
This Web site provides documentation and tutorials for IBM Tivoli monitoring solutions.

IBM Tivoli Monitoring for Web Infrastructure
IBM Tivoli Monitoring for Web Infrastructure provides can help you ensure optimal performance and availability of both application servers and their associated Web servers. See this Web site for more information.

Performance resources

These resources provide additional information about performance:

- WebSphere Application Server performance resources (page 5)
- Java(TM) performance resources (page 6)
- iSeries performance resources (page 6)
- Other performance resources (page 8)

WebSphere Application Server performance resources

WebSphere Application Server 5.0 for iSeries Performance Considerations
This page provides links to information about basic performance considerations for WebSphere Application Server.

WebSphere and Java tuning tips
This page provides links to papers and articles that can help you take advantage of the latest iSeries performance improvements, tools, and tuning methods to optimize WebSphere Application Server performance.

Tuning the WebSphere Prepared Statement Cache
This PDF manual describes how to configure the prepared statement cache in WebSphere Application Server. The prepared statement cache can improve the performance of data access.

WebSphere Performance Management Business Partner Solution
This page provides links to IBM business partners who offer performance monitoring tools that you can use with WebSphere Application Server.

WebSphere Application Server Performance Tuning and Analysis Tools
This high-level online course is offered through IBM eServer Solutions Enablement. The course describes elements that can affect WebSphere Application Server performance, provides basic guidelines to help you acheive optimal performance, and discusses various available tools that you can use to analyze and tune performance.

Java and WebSphere Performance on IBM eServer iSeries Servers
This Redbook provides tips, techniques, and methodologies for working with Java and WebSphere Application Server performance. The specific performance measurements this Redbook are based on versions 3.5 and 4.0 of WebSphere Application Server and OS/400 V5R1. However, the document might provide useful information about general performance concepts and techniques.

Java(TM) performance resources

Tuning Garbage Collection for Java(TM) and WebSphere on iSeries
This PDF manual describes how to configure garbage collection for Java applications that run in WebSphere Application Server. You can find additional information about garbage collection at IBM developerWorks.

Development Best Practices for Performance and Scalability
This white paper describes development best practices for Web applications with servlets, JSP files, JDBC connections, and enterprise applications with EJB components.

Performance Documentation for the Java Platform
Sun Microsystems, Inc. provides links to performance documentation for the Java Platform.

Optimize your Java application's performance
This article examines the optimization process as a whole, rather than focusing on a single technique. You can also use the search function of the IBM developerWorks Web site to find additional information about Java application performance.

Basic Java Performance for iSeries
This white paper explains JIT-MMI, the user classloader verification cache, and memory pool considerations.

iSeries performance resources

Tune Java program performance with the IBM Developer Kit for Java
This Information Center topic describes how you can use the IBM Developer Kit for Java to optimize the performance of your Java applications.

Performance management
This Web site provides extensive information about iSeries performance management.

Performance Management Resource Library
This page provides links to information that you can use to optimize server performance. The resource library includes white papers, articles, tools documentation, and more.

Performance Capabilities Reference Manual
The Performance Management Library provides links to several editions of the Performance Capabilities Reference Manual. This manual includes information about optimizing performance for DB2 UDB for iSeries, Web servers and WebSphere products, and Java applications.

Performance
This Information Center topic provides extensive information about managing and tuning the performance of your iSeries server.

Applications for performance management
This Information Center topic provides links to several tools that you can use to monitor and manage iSeries performance.

Collection Services
You can use Collection Services to collect performance data, which you can analyze with other performance tools. See this Information Center topic for more information.

Performance Tools for iSeries
Performance Tools for iSeries is a set of tools and commands that you can use to view and analyze iSeries performance data in several ways. See this Information Center topic for more information.

Performance Explorer (PEX) Information Center topic
Performance explorer is a data collection tool that helps you identify the causes of performance problems that cannot be identified with the other available tools or general trend analysis. See this Information Center topic for more information.

Performance Explorer (PEX) Web site
This Web site provides additional information about PEX.

Performance Trace Data Visualizer
Performance Trace Data Visualizer (PTDV) for iSeries is a tool for processing, analyzing, and viewing Performance Explorer collection data residing in PEX database files.

Performance Data Collector tool
The Performance Data Collector (PDC) tool provides profile information about the programs that run on the iSeries server. See this Information Center topic for more information.

IBM Performance Management for eServer iSeries Information Center topic
IBM Performance Management for iSeries (formerly known as PM/400) uses Collection Services to gather the nonproprietary performance and capacity data from your server and then sends the data to IBM for analysis. See this Information Center topic for more information.

IBM Performance Management for eServer iSeries Web site
This Web site provides additional information about PM eServer iSeries.

iDoctor for iSeries
iDoctor for iSeries is a suite of applications that can help you monitor performance and troubleshoot common problems on your iSeries server.

Other performance resources

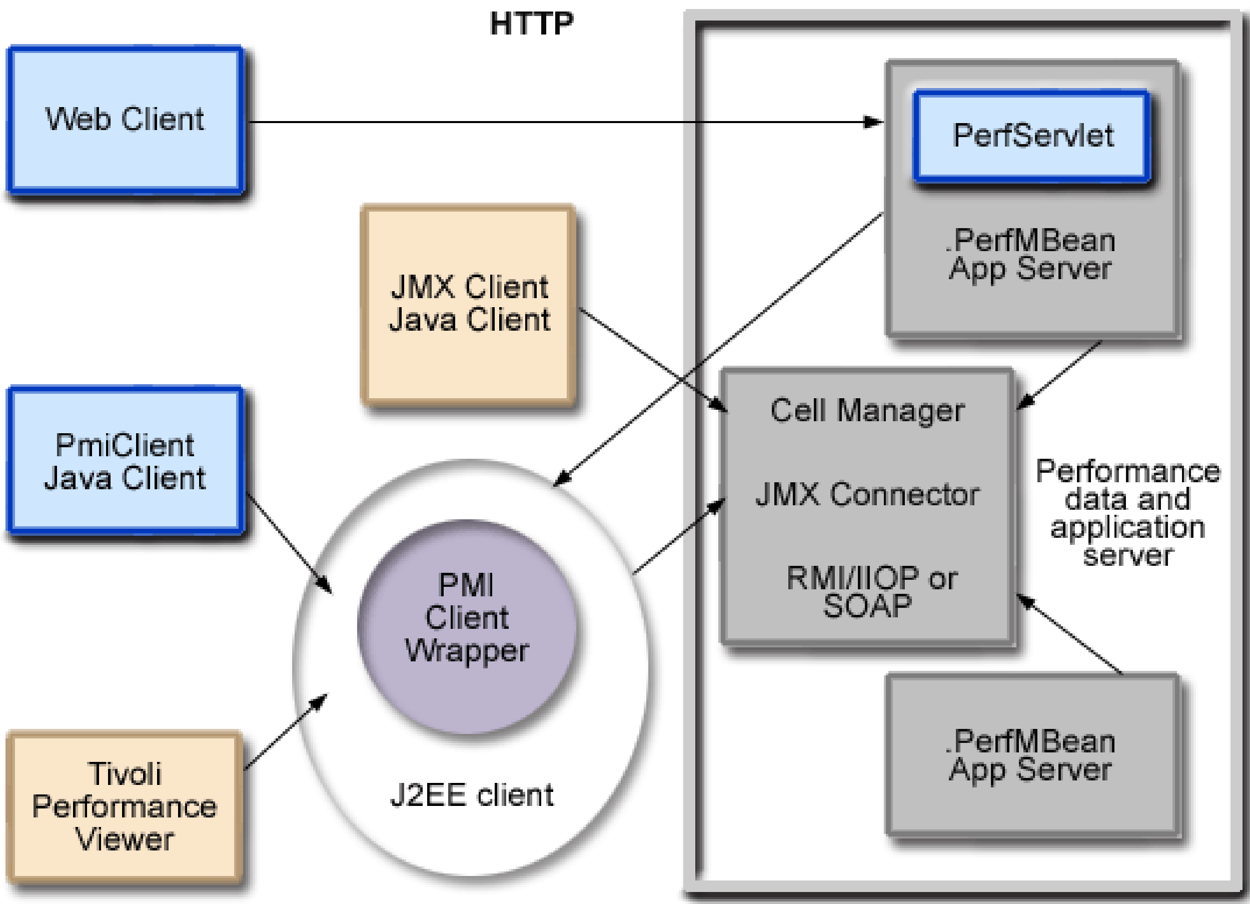
Application Response Measurement (ARM)
This Web site provides information about the Open Group ARM specification.

Tivoli: Performance and Availability Solutions
This Web site provides documentation and tutorials for IBM Tivoli monitoring solutions.

Performance Monitoring Infrastructure (PMI)

The Performance Monitoring Infrastructure (PMI) uses a client-server architecture. The server collects performance data from various WebSphere Application Server components. A client retrieves performance data from one or more servers and processes the data.

As shown in the figure, the server collects PMI data in memory. This data consists of counters such as servlet response time and data connection pool usage. The data points are then retrieved by a Web client, Java client, or JMX client. WebSphere Application Server provides a Java client called Tivoli Performance Viewer. You can also develop custom performance monitoring applications. For more information, see "Develop performance monitoring applications" on page 26.



The figure shows the overall PMI architecture. On the right side, the server updates and keeps PMI data in memory. The left side displays a Web client, a Java client (Tivoli Performance Viewer), and a JMX client retrieving the performance data.

“Performance data organization”
This topic describes the organizational structure of PMI data.

“Performance data classification”
This topic describes the types of performance data that are available.

“PMI data counters” on page 11
This topic provides information about the counters that provide specific performance data.

Performance data organization

PMI data is organized in a centralized hierarchy of these objects:

- **Instance**
  An instance represents a node in the WebSphere Application Server administrative domain. No performance data is collected for the node itself.
- **Server**
  A server is a functional unit that provides services to clients over a network. No performance data is collected for the server itself.
- **Module**
  A module represents one of the resource categories for which collected data is reported to the PMI client. Each module has a configuration file in XML format. This file determines the organization of data and lists a unique identifier for each data counter in the module. Modules include enterprise beans, JDBC connection pools, J2C connection pool, Java Virtual Machine (JVM) run time (including Java Virtual Machine Profiler Interface (JVMPI)), servlet session manager, thread pools, transaction manager, Web applications, Object Request Broker (ORB), Workload Management (WLM), dynamic cache, and Web Services Gateway (WSGW).
- **Submodule**
  A submodule represents a fine granularity of a resource category under the module. For example, ORB thread pool is a submodule of the thread pool category. Submodules can contain other submodules.
- **Counter**
  A counter is a data type that contains performance information for analysis. Each module has an associated set of counters. The data points within a module are queried and distinguished by the Mbean ObjectNames or PerfDescriptors.

You can use Tivoli Performance Viewer to view and manipulate the data for counters. A particular counter type can appear in several modules. For example, both the servlet and enterprise bean modules have a response time counter. In addition, a counter type can have multiple instances within a module.

Performance data classification

Performance Monitoring Infrastructure provides server-side data collection and client-side API to retrieve performance data. Performance data includes two components: static and dynamic.

The static component consists of a name, ID, and other descriptive attributes that identify the data. The dynamic component contains information that changes over time, such as the current value of a counter and the time stamp associated with that value.

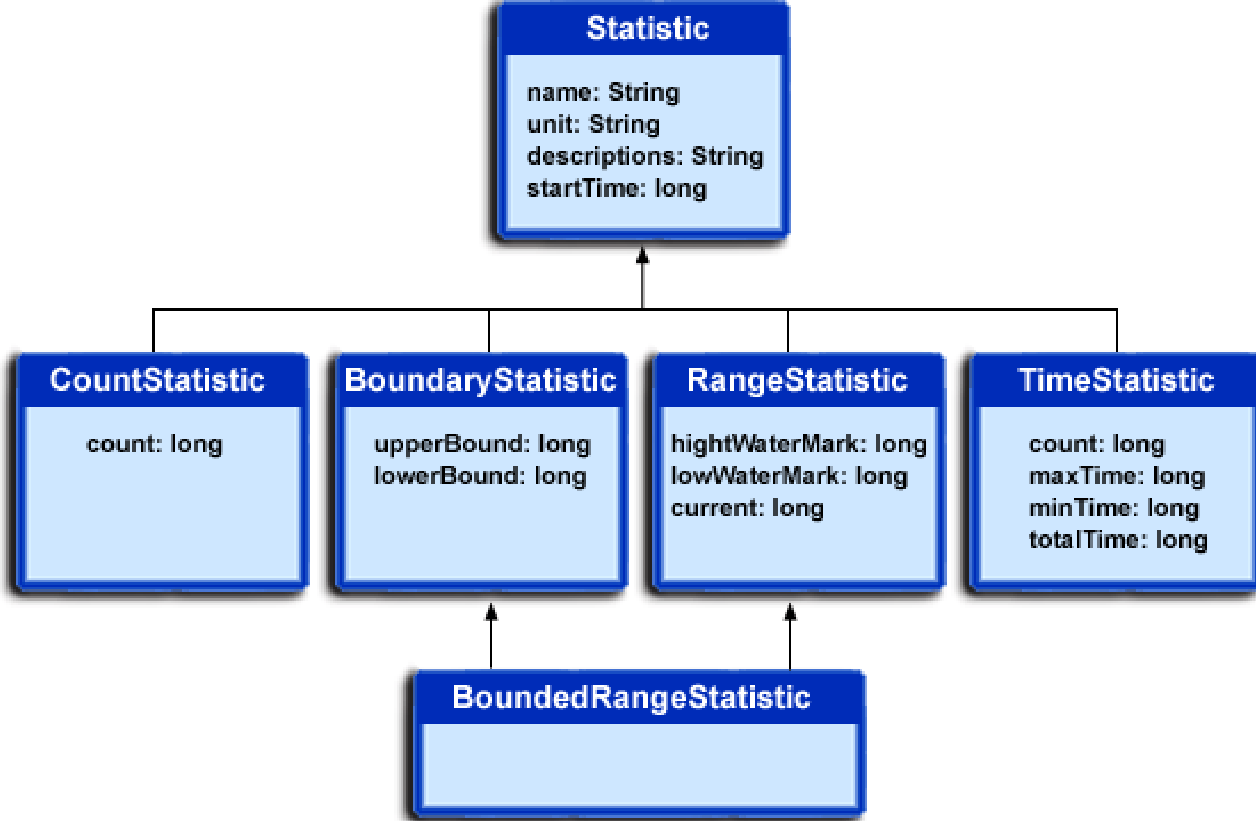
The PMI data can be one of these statistical types defined in the JSR-077 specification:

- CountStatistic
- BoundaryStatistic
- RangeStatistic
- TimeStatistic
- BoundedRangeStatistic

In general, CountStatistic data requires a low monitoring level and TimeStatistic data requires a medium monitoring level. RangeStatistic and BoundedRangeStatistic require a high monitoring level.

There are a few counters that are exceptions to this rule:

- The average method response time, the total method calls, and active methods counters require a high monitoring level.
- The Java Virtual Machine Profiler Interface (JVMPI) counters, SerializableSessObjSize, and data tracked for each individual method (method level data) require a maximum monitoring level.



In previous versions, PMI data was classified with these types:

- **Numeric:** (Corresponds to CountStatistic in the JSR-077 specification.) The Numeric data type contains a single numeric value that can be a long or a double. This data type is used to keep track of simple numeric data, such as counts.
- **Stat:** The Stat data type statistical data on a sample space, including the number of elements in the sample set, their sum, and sum of squares. You can obtain the mean, variance, and standard deviation of the mean from this data.
- **Load:** (Corresponds to the RangeStatistic or BoundedRangeStatistic in the JSR-077 specification.) The Load data type tracks a level as a function of time, including the current level, the time that level was reached, and the integral of that level over time. From this data, you can obtain the time-weighted average of that level. For example, this data type is used in the number of active threads and the number of waiters in a queue.

These PMI data types continue to be supported through the PMI API. Statistical data types are supported through both the PMI API and Java Management Extension (JMX) API.

The TimeStatistic type tracks many counter samples and then returns the sum, count, and average of the samples. For example it can provide an average method response time. Given the nature of this statistic type, it is also used to track non-time related counters, such as average read and write size. You can call the getUnit method on the data configuration information to learn the unit for the counter.

To reduce the impact on performance, numeric and stat data are not synchronized, because this data tracks the total and average values. Load data is very sensitive. Therefore, load counters are always synchronized. In addition, when the monitoring level of a module is set to max, all numeric data is also synchronized to guarantee accurate values. However, setting a monitoring level to max can have a negative impact on performance. As a result, it is recommended that you use the max setting only if it is necessary.

PMI data counters

Counters are enabled at the module level and can be enabled or disabled for elements within the module. Each counter has a specified monitoring level: low, medium, high or maximum. Data collection can affect performance of the application server. The impact depends on the number of counters enabled, the type of counters enabled and the monitoring level set for the counters.

Tivoli Performance Viewer collects and reports performance data for these PMI counters:

- **Enterprise bean module —> Enterprise bean —> Methods in the bean**
  Data counters for this category provide information about enterprise beans, enterprise bean methods, and the remote interfaces that enterprise beans use. Some examples of this information are:
  - The average number of active beans
  - The number of times bean data is loaded or written to the database
  - The number of times a method is called
  - The average response time for a method
  - The number of calls that attempt to retrieve an object from a pool
  - The number of times an object is found in a pool

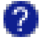
  For more information about these counters, see BeanModule data counters.
- **JDBC connection pools**
  Data counters for this category provide information about connection pools for a database. Some examples of this information are:
  - The average size of the connection pool or number of connections
  - The average number of threads waiting for a connection
  - The average wait time in milliseconds for a connection
  - The average time the connection is in use

  The counters collect performance data for JDBC data sources for versions 4 and 5 of WebSphere Application Server. This information is located under **JDBC Connection Pool** data contents. For a version 4 data source, the data source name is used. For a version 5 data source, the JNDI name is used. For more information about these counters, see JDBC connection pool data counters.
- **J2C connection pool**
  Data counters for this category provide usage information about the Java 2 Enterprise Edition (J2EE) Connector Architecture. This architecture enables managed connections to various resources such as data sources and JMS resources, as well as procedural back-end systems, such as Customer Information Control System (CICS), and Information Management System (IMS). For more information about these counters, see J2C connection pool data counters.
- **Java Virtual Machine API (JVM)**
  Data counters for this category provide information about the memory that a process uses, as reported by Java Virtual Machine (JVM) run time. Some examples of this information are:

– The total memory available

– The amount of free memory for the JVM

The JVM run time also includes data from the Java Machine Profiler Interface (JVMPI). This data provides detailed information about the JVM running the application server. JVMPI is supported on V5R2 and higher. For more information about these counters, see Java Virtual Machine data counters.

 **Notes:**

– The instructions at the top of the help topic do not apply to iSeries. To enable JVMPI, see "Enable Java Virtual Machine Profiler Interface (JVMPI) data reporting" on page 19.

– The total memory counter in the Java Virtual Machine (JVM) data category is a BoundedRangeStatistic type. However, the upperBound and lowerBound are not implemented in WebSphere Application Server Version 5.

– Using JVMPI data can have a significant impact on the performance of your application server. The JVMPI data counters are enabled only when the PMI event JVMRunTimeModule is set to Max.

- **Servlet session manager**
  Data counters for this category provide information about HTTP sessions. Some examples of this information are:
  – The total number of accessed sessions
  – The average amount of time it takes for a session to perform a request
  – The average number of concurrently active HTTP sessions

  For more information about these counters, see Session data counters.
- **Thread pool**
  Data counters for this category provide information about the thread pools for Object Request Broker (ORB) threads and the Web container pools used to process HTTP requests. Some examples of this information are:
  – The number of threads that are created and destroyed
  – The maximum number of pooled threads that are allowed
  – The average number of active threads in the pool

  For more information about these counters, see ThreadPool data counters.
- **Java Transaction API (JTA)**
  Data counters for this category provide information about the transaction manager. Some examples of this information are:
  – The average number of active transactions
  – The average duration of transactions
  – The average number of methods per transaction

  For more information about these counters, see Transaction data counters.
- **Web applications —> Servlet**
  Data counters for this category provide information about the selected server. Some examples of this information are:
  – The number of loaded servlets
  – The average response time for completed requests
  – The number of requests for the servlet

  For more information about these counters, see Web application data counters.
- **Object Request Broker (ORB)**
  Data counters for this category provide information about the ORB. Some examples of this information are:

- The object reference lookup time
- The total number of requests
- The processing time for each interceptor

For more information about these counters, see Object Request Broker data counters.

- **Web Services Gateway (WSGW)**
  Data counters for this category provide information about WSGW. Some examples of this information are:
  - The number of synchronous requests and responses
  - The number of asynchronous requests and responses

  For more information about these counters, see Web service gateway (WSGW) data counters.
- **System data**
  Data counters for this category provide information about a machine (node), such as CPU utilization and memory usage. Due to the design of iSeries architecture, there is no direct mapping of free system memory. Thus, the value of system free memory returns a value zero. Note that this category is available at node level, which means it is only availabe at NodeAgent in WebSphere Application Server Network Deployment. For more information about these counters, see System data counters.
- **Workload Management (WLM)**
  Data counters for this category provide information about workload management. Some examples of this information are:
  - The number of requests
  - The number of updates
  - The average response time

  For more information about these counters, see Workload Management data counters.
- **Dynamic cache**
  Data counters for this category provide information about the dynamic cache service. Some examples of this information are:
  - The memory cache size
  - The number of invalidations
  - The number of hits and misses

  For more information about these counters, see Dynamic cache data counters.
- **Web services**
  Data counters for this category contain information for Web services. Some examples of this information are:
  - The number of loaded Web services
  - The number of requests delivered and processed
  - The request response time
  - The average size of requests

  For more information about these counters, see Web services data counters.

Monitor performance with Tivoli Performance Viewer

Tivoli Performance Viewer (formerly known as Resource Analyzer) is a stand-alone Java performance monitoring client for WebSphere Application Server. You can use Tivoli Performance Viewer to retrieve performance data from application servers. The application servers collect data continuously, and Tivoli Performance Viewer retrieves the data as needed.

Tivoli Performance Viewer provides access to performance data for two kinds of resources:

- Application resources, such as enterprise beans and servlets
- WebSphere Application Server run-time resources, such as Java(TM) virtual machine (JVM) memory, application server thread pools, and database connection pools

Performance data includes simple counters, statistical data (such as the response time for each method invocation of an enterprise bean), and load data (such as the average size of a database connection pool during a specified time interval). This data is reported for individual resources and for multiple resources.

You can use Tivoli Performance Viewer to perform these tasks:

- View data in real time or view historical data from log files.
- View data in chart form, allowing comparisons of one or more statistical values for a given resource on the same chart. You can also specify different units of measurement to enable meaningful graphic displays.
- Record current performance data in a log and replay performance data from previous sessions.
- Compare data for a single resource to a group of resources on a single node.

Install Tivoli Performance Viewer from the WebSphere Application Server workstation CD-ROM for your workstation platform. For more information, see Install the workstation tools for WebSphere Application Server in *Installation*.

These topics provide information about how you can use Tivoli Performance Viewer to monitor your applications running in the WebSphere Application Server environment.

"Start Tivoli Performance Viewer"
This topic describes how to start Tivoli Performance Viewer.

"Set performance monitoring levels" on page 16
This topic describes how to set monitoring levels for Tivoli Performance Viewer.

"View performance data" on page 16
Tivoli Performance Data can display data in different formats. This topic describes how to view performance data in Tivoli Performance Viewer.

"Save performance data to a log file" on page 18
This topic describes how to use Tivoli Performance Viewer to save performance data to a log file.

"Enable Java Virtual Machine Profiler Interface (JVMPI) data reporting" on page 19
This topic describes how to enable JVMPI reporting for the Java virtual machine that runs your application server.

For more information about WebSphere Application Server performance, see Performance Considerations .

Start Tivoli Performance Viewer

Before you start Tivoli Performance Viewer, you must enable performance monitoring services. See these topics for more information:

- "Enable performance monitoring services for application servers" on page 20
- "Enable performance monitoring services for node agents" on page 20 (for Network Deployment only)

Use one of these methods to start Tivoli Performance Viewer:

- On your workstation, click **Start —> Programs —> IBM WebSphere —> Application Server v5.0 —> Tivoli Performance Viewer**. Tivoli Performance Viewer uses the settings in tperfviewer.bat to connect to your application server. If you want to change these settings, see Change the default settings in tperfviewer.bat.
- You can also start Tivoli Performance Viewer from a command prompt:
  1. Open a command prompt.
  2. Run the `cd` command to change to the *product_installation_directory*\bin directory.
  3. Run the tperfviewer script:

     ```
     tperfviewer.bat host_name [ port_number ] [ connector_type ]
     ```

     where *host_name* is the iSeries host name, *port_number* is the SOAP or RMI connector port for the application server from which you are collecting data, and *connector_type* is the type of connector to use. The default value for *port_number* is 8880 for WebSphere Application Server and 8879 for WebSphere Application Server Network Deployment. Valid values for *connector_type* are SOAP or RMI. The default connector type is SOAP.

     Note: You can change the default values in tperfviewer.bat so that you do not need to specify the host name or port number when you run the tperfviewer.bat script. See Change the default settings in tperfviewer.bat for more information.

If you plan to run Tivoli Performance Viewer in a secured environment, see Run monitoring applications with security enabled (page 42).

After you start Tivoli Performance Viewer, you can perform these tasks:
- "Set performance monitoring levels" on page 16 for resources
- "View performance data" on page 16 based on the current settings
- "Save performance data to a log file" on page 18

Change the default settings in tperfviewer.bat

You can change the default settings in the tperfviewer.bat file so that Tivoli Performance Viewer connects to your application server. To change the settings, follow these steps:
1. On your workstation, open C:*product_installation_directory*\bin\tperfviewer.bat in a text editor. The default installation directory is C:\Program Files\WebSphere\AppServer.
2. Under the **:LOCAL** entry, replace these lines:

   ```
   set DEST=localhost
   set DESTPORT=8879
   ```

   with these lines:

   ```
   set DEST=host_name
   set DESTPORT=port_number
   ```

   where *host_name* is the name of your iSeries server and *port_number* is the SOAP port for your application server.

Note: These instructions assume you are using the SOAP connector. If you use the RMI connector, specify the application server's RMI port number and replace this line:

```
set CONNECTOR=SOAP
```

with this one:

```
set CONNECTOR=RMI
```

Set performance monitoring levels

The monitoring settings determine which counters are enabled. Changes made to the settings from Tivoli Performance Viewer affect all applications that using the Performance Monitoring Infrastructure (PMI) data.

To change the monitoring levels for enterprise bean methods, see Set monitoring levels for enterprise bean methods (page 16).

To set monitoring settings, follow these steps:

1. Click **Data Collection** in the navigation tree.
2. Select one of these monitoring levels for your application server:
   - **None**: Provides no data collection
   - **Standard**: Enables data collection for all modules except enterprise bean method level data
   - **Custom**: Allows customized settings for each module
3. (Optional) Fine tune the monitoring level settings. If you select the **Custom** monitoring setting, you can specify settings for each module.
   a. Click **Specify**.
   b. For each resource, select one of these monitoring levels:
      - None
      - Low
      - Medium
      - High
      - Maximum

   The window displays the counters that are enabled for each resource based on the monitoring level that you specify.

   Note: When you set the monitoring level for a resource, the same level is set recursively for that resource's child resources. To override the setting for a child resource, set its level after you set the level for the parent resource.
4. Click **OK**.
5. Click **Apply**.

Set monitoring levels for enterprise bean methods

Because of the impact on performance, the **Standard** monitoring level does not include monitoring for individual remote methods.

To configure monitoring for individual methods, follow these steps:

1. Click **Data Collection** in the navigation tree.
2. In the **Current Activity Settings**, select **Custom**.
3. Click **Specify**.
4. Expand **Enterprise Beans —>** ***application_name*** **—>** ***bean_type*** **—>** ***bean_name*** **—> Methods**.
5. Set the monitoring level to **Maximum**.
6. Click **OK**.
7. Click **Apply**.

View performance data

Tivoli Performance Viewer presents performance data in two formats: a summary report and a chart.

The summary reports display a summary of all performance data for an application server. The charts display real-time performance data for specific resources.

These topics describe how to display performance data in Tivoli Performance Viewer:

- View summary reports for an application server (page 17)
- View chart data for resources (page 17)
- Performance data refresh behavior (page 17)

View summary reports for an application server

Before you view the summary report, verify that data counters are enabled and monitoring levels are set properly.

The standard monitoring level enables all reports except the report on enterprise bean methods. To enable enterprise bean method reports, use the custom monitoring setting and set the monitoring level to Max for the enterprise bean module.

To view the summary reports, follow these steps:

1. In the navigation tree, click the icon for your application server.
2. Select the tab for the summary report that you want to view.
3. To sort the data based on a data counter, click the column header for the counter.

View data for resources

The View Data tab displays a performance data in a report format. To view data in this format, follow these steps:

1. Click a resource in the resource selection panel.
2. Click the **View Data** tab in the data monitoring panel.

The View Chart tab displays a graph with time as the x-axis and the performance value as the y-axis. To view data in this format, follow these steps:

1. Click a resource in the resource selection panel.
2. Click the **View Chart** tab in the data monitoring panel.

Negative results display as zero (0). If necessary, you can type a scaling factor in the **Scale** field for each counter.

Performance data refresh behavior

New performance data becomes available in these situations:

- You change the monitoring level for a resource.
- You add a new resource or application to the run time.

Tivoli Performance Viewer handles these changes in the following ways:

- If Tivoli Performance Viewer is monitoring the new or changed resource, or the parent of the resource, the system is automatically refreshed.
- If you add counters for a group that the performance viewer is monitoring, the performance viewer automatically adds the counters to the table and chart views.
- If Tivoli Performance Viewer is monitoring the parent of a new resource, the new resource is detected automatically and added to the Resource Selection tree.

To manually refresh the Resource Selection tree, or parts of it, select the appropriate node and click the **Refresh** icon, or right-click a resource and select **Refresh**.

When an application server runs, the performance viewer tree automatically updates the server local structure, including its containers and enterprise beans, to reflect changes on the server. However, if a stopped server starts *after* the performance viewer starts, a manual refresh operation is required so that the server structure accurately reflects in the Resource Selection tree.

For more information on configuring and using Tivoli Performance Viewer, see these help topics:

- Changing the refresh rate of data retrieval
- Changing the display buffer size
- Viewing and modifying performance chart data
- Scaling the performance data chart display
- Clearing values from tables and charts
- Resetting the counters to zero
- Replaying a performance data log file

Save performance data to a log file

You can save all performance data from Tivoli Performance Viewer in a log file and write the data in binary format (serialized Java objects) or XML format.

To enable logging, follow these steps:

1. Click **Logging —> On** or click the Logging icon.
2. Specify a name for the log file. You can save the log file in binary format with the PERF file extension, or in XML format. The XML files are compatible with vendor software.

   Note: The *.perf files may not be compatible between fix levels of WebSphere Application Server.
3. Click **OK**.

To disable logging click **Logging —> Off** or click the Logging icon.

A log file is only generated if you enable logging. By default, this data is written to this file on your workstation:

```
C:\product_installation_directory\logs\tpv_mmdd_hhmm.xml
```

where *mmdd* is the month and date, and *hhmm* is the hour and minute.

The log file is not updated, but remains available for you to replay the collected data. For information on how to replay a log file, see Replaying a performance data log file. Replaying the performance data log file does not have an effect on the WebSphere Application Server environment.

Example

This is an example of an XML log file:

```
<?xml version="1.0"?>
<RALog version="5.0">
  <RAGroupSnapshot time="1019743202343" numberGroups="1">
    <CpdCollection name="root/myDir/Default Server/jvmRuntimeModule"
        level="7">
      <CpdData name="root/myDir/Default Server/jvmRuntimeModule/jvmRuntimeModule.total/Memory"
```

```
        id="1">
      <CpdLong value="39385600" time="1.019743203334E12"/>
    </CpdData>
    <CpdData name="root/myDir/Default Server/jvmRuntimeModule/jvmRuntimeModule.freeMemory"
        id="2">
      <CpdLong value="4815656" time="1.019743203334E12"/>
    </CpdData>
    <CpdData name="root/myDir/Default Server/jvmRuntimeModule/jvmRuntimeModule.usedMemory"
        id="3">
      <CpdLong value="34569944" time="1.019743203334E12"/>
    </CpdData>
  </CpdCollection>
 </RAGroupSnapshot>
</RALog>
```


Enable Java Virtual Machine Profiler Interface (JVMPI) data reporting

Tivoli Performance Viewer uses a Java Virtual Machine Profiler Interface (JVMPI) to enable more comprehensive performance analysis. This profiling tool enables the collection of information such as data about garbage collection for the Java virtual machine (JVM) that runs the application server.

JVMPI is a two-way function call interface between the JVM API and an in-process profiler agent. The JVM API notifies the profiler agent of various events, such as heap allocations and thread starts. The profiler agent can activate or deactivate specific event notifications, based on the needs of the profiler.

JVMPI supports partial profiling. You can select the types of profiling information to collect and select certain subsets of the time during which the JVM API is active.

Note: JVMPI collection is resource intensive and adversely affects the performance of your application server. It is recommended that you only use JVMPI for short periods of time to debug specific problems.

Enable Java Virtual Machine Profiler Interface data reporting

To enable Java Virtual Machine Profiler Interface (JVMPI) data reporting for an individual application server, perform these steps:

1. “Enable performance monitoring services for application servers” on page 20. If you are running WebSphere Application Server Network Deployment, you must also “Enable performance monitoring services for node agents” on page 20.
2. Start the administrative console.
3. In the topology tree, expand **Servers** and click **Application Servers**.
4. Click the application server for which you want to enable JVMPI.
5. Click **Process Definition**.
6. Click **Java Virtual Machine**.
7. Type *-XrunQEJBJVMPI* in the **Generic JVM arguments** field.
8. Click **Apply** or **OK**.
9. Save the configuration.
10. Start the application server, or restart the application server if it is currently running.
11. In Tivoli Performance Viewer, perform these steps:
    - Expand **Data Collection**.
    - Select **Specify**.
    - Select **JVM Runtime** and set **Monitoring Level** to **Maximum**.
    - Click **OK**.
    - Click **Apply**.

Enable performance monitoring services

By default, performance monitoring services are disabled. These topics describe how to enable performance monitoring and data collection:

"Enable performance monitoring services for application servers"
This topic describes how to use the administrative console to enable performance monitoring for an application server.

"Enable performance monitoring services for node agents"
This topic describes how to use the administrative console to enable performance monitoring for a node agent in WebSphere Application Server Network Deployment.

"Enable performance monitoring services with wsadmin" on page 21
This topic describes how to use wsadmin to collect performance data with MBeans.

"Enable performance data collection with the administrative console" on page 22
This topic describes how to use the administrative console to enable data collection for PMI modules and to set monitoring levels.

Enable performance monitoring services for application servers

By default, performance monitoring is disabled for application servers. To permanently enable the performance monitoring services for an application server, perform these steps:

1. Start the administrative console.
2. In the topology tree, expand **Servers** and click **Application Servers**.
3. Click the name of the server for which you want to enable performance monitoring.
4. Click **Performance Monitoring Service**.
5. Select **Startup**.
6. Specify the **Initial specification level**. Options are None, Standard or Custom. If you specify None, you can use Tivoli Performance Viewer to adjust the levels after you restart the application server.
7. Click **Apply** or **OK**.
8. Save the configuration.
9. Restart the application server.

To dynamically enable the performance monitoring services for an application server, after restarting your application server, enable the trace levels using Tivoli Performance Viewer to the specific tracing levels. For more information, see "Set performance monitoring levels" on page 16.

For more information on the configuration settings, see Performance monitoring service settings.

Enable performance monitoring services for node agents

By default, performance monitoring is disabled for node agents. To enable performance monitoring services for a node agent, perform these steps:

1. Start the administrative console.
2. In the topology tree, expand **System Administration** and click **Node Agents**.
3. Click the name of the node agent for which you want to enable performance monitoring.
4. Click **Performance Monitoring Service**.
5. Select the check box in the **Startup** field.
6. Click **Apply** or **OK**.
7. Save the configuration.

8. Restart the node agent.

To dynamically enable the performance monitoring services for an application server, after restarting your application server, enable the trace levels using Tivoli Performance Viewer to the specific tracing levels. For more information, see "Set performance monitoring levels" on page 16.

For more information on the configuration settings, see Performance monitoring service settings.

Enable performance monitoring services with wsadmin

You can use wsadmin to invoke operations on PerfMbean to obtain PMI data, set or obtain PMI monitoring levels, and enable data counters. For more information about wsadmin, see The wsadmin administrative tool.

The following operations in PerfMBean can be used in wsadmin:

```
/** Set instrumentation level using String format
*  This should be used by scripting for an easy String processing
*/
public void setInstrumentationLevel(String levelStr, Boolean recursive);

/** Get instrumentation level in String for all the top level modules
*  This should be used by scripting for an easy String processing
*/     public
String getInstrumentationLevelString();

 /** Return the PMI data in String
*
*/     public
String getStatsString(ObjectName on, Boolean recursive);

/** Return the PMI data in String
*  Used for PMI modules/submodules without direct MBean mappings.
*/     public
String getStatsString(ObjectName on, String submoduleName, Boolean recursive);

/**
* Return the submodule names if any for the MBean
*/
public String listStatMemberNames(ObjectName on);
```

If an MBean is a StatisticProvider and if you pass its ObjectName to getStatsString, you get the Statistic data for that MBean. MBeans with the following MBean types are statistic providers:

- DynaCache
- EJBModule
- EntityBean
- JDBCProvider
- J2CResourceAdapter
- JVM
- MessageDrivenBean
- ORB
- Server
- SessionManager
- StatefulSessionBean
- StatelessSessionBean
- SystemMetrics
- ThreadPool

- TransactionService
- WebModule
- Servlet
- WLMAppServer
- WebServicesService
- WSGW

Enable performance data collection with the administrative console

To enable data collection in the administrative console, select the Performance Monitoring Infrastructure (PMI) modules and levels that you want to monitor.

1. Start the administrative console.
2. In the topology tree, expand **Servers** and click **Application Servers**.
3. Click the name of the server for which you want to enable data collection.
4. Click the **Runtime** tab.
5. Click **Performance Monitoring Service**.
6. Select the PMI modules and levels to set the initial specification level field.
7. Click **Apply** or **OK**.
8. Save the configuration.

These changes take effect immediately, but are not persistent. Use the Configuration tab for a persistent change. See “Enable performance monitoring services for application servers” on page 20 for more information about making a persistent change.

Measure data requests with PMI Request Metrics

Performance Monitoring Infrastructure (PMI) Request Metrics measures the time that it takes for requests to travel through WebSphere Application Server components. This data helps to identify run time and application problems. PMI Request Metrics measures how long a request stays at points such as the Web container, the enterprise bean container, and the database. This information is recorded in logs, and can be written to Application Response Measurement (ARM) agents used by Tivoli monitoring tools.

To record data with PMI Request Metrics, perform these steps:

1. “Enable PMI Request Metrics” on page 23.
2. (Optional) “Enable Application Response Measurement” on page 23.
3. (Optional) “Configure Request Metrics trace filters” on page 24. Filters specify which requests you want to trace.
4. (Optional) “Set the trace level for Request Metrics” on page 25. The trace level specifies how much information you want to collect for incoming requests.
5. Regenerate the Web server plugin configuration. After you modify the Request Metrics configuration, you must regenerate the Web server plug-in configuration file so that the Web server plug-in recognizes the changes to the Request Metrics configuration. If you make multiple changes to Request Metrics, regenerate the plug-in configuration files after you complete all changes. If you do not regenerate the plug-in configuration, the Web server plug-in might have different Request Metrics configuration data from the application server. This difference in configuration data might cause inconsistent behaviors for request metrics between the Web server plug-in and the application server.

See this topic for information about the trace output file:

“Request Metrics data output” on page 25
This topic describes the format of the Request Metrics trace output.

Enable PMI Request Metrics

PMI Request Metrics measures process hop response times in multi-tiered applications and records the data in system logs. You can use this data to identify run-time and application performance problems.

For processes that start from HTTP or enterprise bean remote requests, Request Metrics measures response times for the initiating request and any related downstream enterprise bean invocations and Java Database Connectivity (JDBC) calls. Request Metrics also provides the same information on process hop response time through the Application Response Measurement (ARM) interface.

Request Metrics compares each incoming request to a set of filters. If the request matches any filter with a trace level greater than TRACE_NONE, trace records are generated for that request.

Typically, requests enter the system and create processes that are distributed across several nodes within a distributed system. Each process can be further distributed and call other processes. When the processes are distributed, trace records are generated for each process. These trace records can be correlated together to build a sequence diagram of the response times for the request. The processes are only recorded if they are generated through a remote enterprise bean call.

You can enable Request Metrics without enabling Application Response Measurement (ARM).

To enable Request Metrics, perform these steps:

1. Start the administrative console.
2. In the administrative console navigation tree, click **Troubleshooting**.
3. Click **PMI Request Metrics**.
4. Select the check box in the **Enable** field under the **Configuration** tab.
5. Click **Apply** or **OK**.
6. Save the configuration.
7. Stop and restart the application server.

Enable Application Response Measurement

Application Response Measurement (ARM) is an Open Group standard composed of a set of interfaces. An ARM agent implements these interfaces to provide information on elapsed time for process hops.

Before you enable Application Response Measurement (ARM), install a supported ARM implementation on all WebSphere Application Server nodes. WebSphere Application Server does not provide an ARM implementation. Verify with your ARM agent provider that Request Metrics is supported by the ARM agent implementation. ARM support is dependent on Request Metrics support.

Note: Request Metrics in the Web server plug-in is not integrated with ARM in WebSphere Application Server Version 5.0.x. If ARM is enabled, Request Metrics in the Web server plug-in ignores it.

1. Install the ARM implementation. Add this line to the startup command for the application servers:
   ```
   -Dcom.ibm.websphere.pmi.reqmetrics.ARMIMPL=ARMIMPLNAME
   ```
   WebSphere Application Server support of ARM is dependent on Request Metrics support. If enabled, and an appropriate ARM implementation is defined to the server run times, then the ARM implementation is called as requests enter WebSphere Application Server processes and when Java Database Connectivity (JDBC) calls are made, using EJB 2.0 data sources.
2. Start the administrative console.
3. Expand **Troubleshooting** and click **PMI Request Metrics**.
4. On the **Performance Monitoring Request Metrics** page, select the **enableARM** check box.
5. Save the configuration.
6. Restart the application server.

Configure Request Metrics trace filters

To specify which requests you want to trace with Request Metrics, set trace filters. The data is recorded to the system log file standard output and can be used for real-time and historical analysis.

If you collect request metrics data in a production environment, it is recommended that you filter by IP address. If you choose to enable Request Metrics, but not filter by a specific IP address, performance can be adversely affected. If you do not want to filter by IP address, it is recommended that you enable Request Metrics collection in test environments only.

Incoming HTTP requests

You can filter incoming HTTP requests based on client IP address and the URI of the request.

- **Client IP address filters.** Requests are filtered based on a known IP address. Use an asterisk (*) to specify a mask for an IP address. The asterisk must always be the last character of the mask. For example, 127.0.0.* is a valid filter, but 127.0.*.30 is not. For performance reasons, the pattern matches character by character, until either an asterisk is found in the filter, a mismatch occurs, or the filters are found as an exact match.
- **URI filters.** Requests are filtered based on the URI of the incoming HTTP request. URI filters and client IP address filters use the same pattern matching rules.
- **Filter combinations.** If both URI and client IP address filters are active, then Request Metrics requires a match for both filter types. If neither filter type is active, all requests are considered a match.

Incoming enterprise bean requests

You can filter enterprise bean requests based on the full name of the enterprise bean method. As with IP address and URI filters, you can use the asterisk (*) to provide a mask. The asterisk must always be the last character of a filter pattern.

Enable Request Metrics filters

To enable Request Metrics filters, follow these steps:

1. Start the administrative console.
2. In the administrative console navigation tree, click **Troubleshooting**.
3. Click **PMI Request Metrics**.
4. Click **Filters**.
5. Click the filter type that you want to enable.
6. Select the check box in the **Enable** field under the **Configuration** tab.
7. Click **Apply** or **OK**.
8. Save the configuration.
9. Stop and restart the application server.

Note: You can enable or disable a filter group. If the group is enabled, you can enable or disable individual filters within the group.

Add and remove Request Metrics filters

To configure Request Metrics filters, follow these steps:

1. Start the administrative console.
2. In the administrative console navigation tree, click **Troubleshooting**.
3. Click **PMI Request Metrics**.
4. Click **Filters**.

5. Click the filter type that you want to configure.
6. Click **New**.
7. Choose a filter type from the drop down box in the type field under the **Configuration** tab.
8. Select the check box in the **Enable** field to enable the filter.
9. Click **filterValues**.
10. Make changes to the filter.
11. Click **Apply** or **OK**.
12. Save the configuration.
13. Stop and restart the application server.

For more information, see these help topics:

- Performance Monitoring Infrastructure Request Metrics configuration settings
- PMIRMFilter collection
- PMIRM filter settings
- Filter value collection
- Filter value settings

Set the trace level for Request Metrics

The trace level specifies how much information you want to collect for incoming requests. To set the trace level to generate records, follow these steps:

1. Start the administrative console.
2. In the topology tree, expand **Troubleshooting** and click **PMI Request Metrics**.
3. Select the trace level that you want to use. If the trace level is set to NONE, no records are generated.
4. Click **Apply** or **OK**.
5. Save the configuration.
6. Stop and restart the application server.

Request Metrics data output

Data in the trace record output file has this format:

```
PMRM0003I: parent:ver=n,ip=n.n.n.n,time=nnnnnnnnnn,pid=nnnn,reqid=nnnnnn,event=nnnn
- current:ver=n,ip=n.n.n.n,time=nnnnnnnnnn,pid=nnnn,reqid=nnnnnn,event=nnnn
    type=TTT detail=some_detail_information elapsed=nnnn bytesIn=nnnn
    bytesOut=nnnn
```

The trace record format consists of two correlators: a parent correlator and current correlator. The parent correlator represents the upstream request and the current correlator represents the current operation. If the parent and current correlators are the same, then the record represents an operation that occurred as it entered WebSphere Application Server.

To correlate trace records for a particular request, collect records with a message ID of PMRM0003I from the appropriate server logs. Records are correlated by matching current correlators to parent correlators. The logical tree can be created by connecting the current correlators of parent trace records to the parent correlators of child records. This tree shows the progression of the request across the server cluster.

The parent correlator is denoted by the comma-separated fields that follow the keyword `parent:`. Likewise, the current correlator is denoted by the comma-separated fields that follow `current:`.

The fields of both parent and current correlators are as follows:

- **ver:** The version of the correlator. For convenience, the version is listed in both the parent and current correlators.
- **ip:** The IP address of the application server node that generates the correlator.
- **pid:** The process ID of the application server that generates the correlator.
- **time:** The start time of the application server process that generates the correlator.
- **reqid:** An ID assigned to the request by Request Metrics. This ID is unique to the application server process.
- **event:** An event ID assigned to differentiate the trace events.

The metrics data for the timed operation is listed after the parent and current correlators:
- **type:** A 3 character code that represents the type of operation that is traced. Values include HTTP, URI, EJB, and JDBC.
- **detail:** Identifies the name of the operation.
- **elapsed:** The measured elapsed time in units for this operation, including all sub-operations that this operation calls.
- **bytesIn:** The number of bytes from the request received by the Web server plug-in.
- **bytesOut:** The number of bytes from the reply sent from the Web server plug-in to the client.

The type and detail fields identify the operation:
- **HTTP:** The Web server plug-in generates the trace record. The detail is the name of the URI used to invoke the request.
- **URI:** A Web component generates the trace record. The URI is the name of the URI that invokes the request.
- **Enterprise bean:** The fully qualified package and method name of the enterprise bean.
- **JDBC:** The values select, update, insert or delete for prepared statements. For non-prepared statements, the full statement can appear.

Develop performance monitoring applications

WebSphere Application Server provides PMI interfaces that you can use to develop custom performance monitoring applications. These are the available interfaces:
- The PMI client interface
- The PMI servlet interface
- The Java Management Extensions (JMX) interface

“Develop performance monitoring applications with the PMI client” on page 27
The PMI client interface is a Java interface. This topic describes how to develop performance monitoring applications with the PMI client.

“Develop performance monitoring applications with the PMI servlet” on page 35
The PMI servlet interface returns PMI data in XML format. This topic describes how to develop performance monitoring applications with the PMI servlet.

“Access PMI data with the JMX interface” on page 38
The JMX interface invokes methods on MBeans to access PMI data. This topic describes how to access performance data on one or more MBeans with the JMX interface.

“Run monitoring applications” on page 40
This topic describes how to run a PMI client application.

“Run monitoring applications with security enabled” on page 41
This topic describes how to run a PMI client application in a secured environment.

Develop performance monitoring applications with the PMI client

A Performance Monitoring Infrastructure (PMI) client is an application that receives PMI data from servers and processes this data. You can use the PMI client package to write PMI clients that collect and display PMI data from servers.

Clients can be graphical user interfaces (GUIs) that display performance data in real-time, applications that monitor performance data and trigger different events according to the current values of the data, or any other application that needs to receive and process performance data.

The programming model for PMI clients consists of these steps:

1. Create an instance of PmiClient. This instance is used for all subsequent method calls.
2. Call the listNodes() and listServers(nodeName) methods to find all of the nodes and servers in the WebSphere Application Server domain. The PMI client provides two sets of methods:
   - A set for Version 5
   - A set from Version 4>/li>

   Use only one set of methods.
3. Call listMBeans and listStatMembers to get all of the available MBeans and MBeanStatDescriptors.
4. Call the getStats method to get the Stats object for the PMI data.
5. (Optional) The client can also call setStatLevel or getStatLevel to set and get the monitoring level. Use the MBeanLevelSpec objects to set monitoring levels.

For the WebSphere Application Server Version 4 PMI interface, the object types are different. If you want to use the Version 4 interface, follow these steps:

1. Create an instance of PmiClient.
2. Call the listNodes() and listServers(nodeName) methods to find all of the nodes and servers in the WebSphere Application Server domain.
3. Call listMembers to get all of the perfDescriptor objects.
4. Use the PMI client's `get` or `gets` method to get CpdCollection objects. These objects contain performance data from the server. The same structure is maintained and its update method is used to refresh the data.
5. (Optional) The client can also call setInstrumentationLevel or getInstrumentationLevel to set and get the monitoring level.

See these topics for additional information about the PMI client package.

"The PMI client package"
This topic describes the WebSphere Application Server PMI client package.

"The PMI client interface" on page 28
The PMI client interface provides data in a hierarchical structure. This topic describes that structure.

"Example: PMI client with new data structure" on page 29
This topic provides an example of a PMI client that uses the new data structure.

The PMI client package

The Performance Monitoring Infrastructure (PMI) client package provides a wrapper class PmiClient to deliver PMI data to a client.

PmiClient communicates with the network manager first, retrieving an AdminClient instance for each application server. After the PmiClient receives the instance, it uses it to communicate with the application server directly for performance or level setting changes. Because level settings are persistent through PmiClient, you do not need to reset the levels unless you want to change them.

Performance Monitoring Infrastructure and Java Management Extensions

The PmiClient API does not work if the Java Management Extensions (JMX) infrastructure and Perf MBean are not running. If you prefer to use the AdminClient API directly to retrieve PMI data, you still have a dependency on the JMX infrastructure.

When you use the PmiClient API, you must pass the JMX connector protocol and port number to instantiate an object of the PmiClient. After you get a PmiClient object, you can call its methods to list nodes, servers, and MBeans; set the monitoring level; and retrieve PMI data.

The PmiClient API creates an instance of the AdminClient API and delegates your requests to the AdminClient API. The AdminClient API uses the JMX connector to communicate with the PerfMBean in the corresponding server and then returns the data to the PmiClient, which returns the data to the client.

The PMI client interface

The PMI client interface provides data in a hierarchical structure. Descending from the object are node information objects, module information objects, CpdCollection objects and CpdData objects. The client interface provides Stats and Statistic objects. The node and server information objects contain no performance data, only static information.

Each time a client retrieves performance data from a server, the data is returned in a subset of this structure; the form of the subset depends on the data retrieved. You can update the entire structure with new data, or update only part of the structure.

The JMX statistic data model is supported, as well as the CPD data model from WebSphere Application Server, Version 4.0. When you retrieve performance data with The Version 5 PMI client API returns the Stats object, which includes Statistic objects and optional sub-Stats objects. The Version 4.0 PMI client API to collect performance data returns the CpdCollection object, which includes the CpdData objects and optional sub-CpdCollection objects.

These additional PMI interfaces are supported:

- BoundaryStatistic
- BoundedRangeStatistic
- CountStatistic
- MBeanStatDescriptor
- MBeanLevelSpec
- New Methods in PmiClient
- RangeStatistic
- Statistic
- Stats
- TimeStatistic

These PMI interfaces from WebSphere Application Server Version 4.0 are also supported:

- CpdCollection (This interface corresponds to Stats.)
- CpdData
- CpdEventListener and CpdEvent
- CpdFamily class
- CpdValue
  - CpdLong (This interface corresponds to CountStatistic.)
  - CpdStat (This interface corresponds to TimeStatistic)
  - CpdLoad (This interface corresponds to RangeStatistic and BoundedRangeStatistic.)

- PerfDescriptor
- PmiClient class

Notes:

1. Version 4.0 PmiClient APIs are supported in this version, but there are some changes. The data hierarchy is changed in some PMI modules, such as the enterprise bean module and HTTP sessions module. If want to run an existing PmiClient application with Version 5, you might need to update the PerfDescriptor objects based on the new PMI data hierarchy.
2. In Version 5, the getDataName and getDataId methods in PmiClient are non-static methods and support multiple WebSphere Application Server versions. It is recommended that you update your existing application if it uses these two methods.

Example: PMI client with new data structure

This example code uses Performance Monitoring Infrastructure (PMI) client with the new data structure.

Note: Read the “Code example disclaimer” on page 81 for important legal information.

```
import com.ibm.websphere.pmi.*;
import com.ibm.websphere.pmi.stat.*;
import com.ibm.websphere.pmi.client.*;
import com.ibm.websphere.management.*;
import com.ibm.websphere.management.exception.*;
import java.util.*;
import javax.management.*;
import java.io.*;

/**
 * Sample code to use PmiClient API (new JMX-based API in 5.0) and
get Statistic/Stats objects.
 */

public class PmiClientTest implements PmiConstants {

    static PmiClient pmiClnt = null;
    static String nodeName = null;
    static String serverName = null;
    static String portNumber = null;
    static String connectorType = null;
    static boolean success = true;


    /**
     * @param args[0] host
     * @param args[1] portNumber, optional, default is 2809
     * @param args[2] connectorType, optional, default is RMI connector
     * @param args[3]serverName, optional, default is the first server found
     */
    public static void main(String[] args) {

        try {

            if(args.length > 1) {
                System.out.println("Parameters: host [portNumber]
[connectorType] [serverName]");
                return;
            }

            // parse arguments and create an instance of PmiClient
            nodeName = args[0];

            if (args.length > 1)
            portNumber = args[1];

            if (args.length > 2)
```

```
connectorType = args[2];

// create an PmiClient object
pmiClnt = new PmiClient(nodeName, portNumber, "WAS50", false, connectorType);

// Uncomment it if you want to debug any problem
//pmiClnt.setDebug(true);

// update nodeName to be the real host name
nodeName = pmiClnt.getConnectedHost();
System.out.println("use node " + nodeName);

if (args.length == 4)
    serverName = args[3];
else { // find the server you want to get PMI data
    // get all servers on this node
    PerfDescriptor[] allservers = pmiClnt.listServers(nodeName);
    if (allservers == null || allservers.length == 0) {
        System.out.println("No server is found on node " + nodeName);
        System.exit(1);
    }

    // get the first server on the list. You may want to get a different server
    serverName = allservers[0].getName();
    System.out.println("Choose server " + serverName);
}

// get all MBeans
ObjectName[] onames = pmiClnt.listMBeans(nodeName, serverName);

// Cache the MBeans we are interested
ObjectName perfOName = null;
ObjectName serverOName = null;
ObjectName wlmOName = null;
ObjectName ejbOName = null;
ObjectName jvmOName = null;
ArrayList myObjectNames = new ArrayList(10);

// get the MBeans we are interested in
if(onames != null) {
    System.out.println("Number of MBeans retrieved= " + onames.length);
    AttributeList al;
    ObjectName on;
    for(int i=0; i<onames.length; i++) {
        on = onames[i];
        String type = on.getKeyProperty("type");

        // make sure PerfMBean is there.
        // Then randomly pick up some MBeans for the test purpose
        if(type != null &&type.equals("Server"))
            serverOName = on;
        else if(type != null &&type.equals("Perf"))
            perfOName = on;
        else if(type != null &&type.equals("WLM")) {
            wlmOName = on;
        }
        else if(type != null &&type.equals("EntityBean")) {
            ejbOName = on;

            // add all the EntityBeans to myObjectNames
            myObjectNames.add(ejbOName);  // add to the list
        }
        else if(type != null &&type.equals("JVM")) {
            jvmOName = on;
        }
    }
```

```
            // set monitoring level for SERVER MBean
            testSetLevel(serverOName);

            // get Stats objects
            testGetStats(myObjectNames);

            // if you know the ObjectName(s)
            testGetStats2(new ObjectName[]{jvmOName, ejbOName});

            // assume you are only interested in a server data in WLM MBean,
            // then you will need to use StatDescriptor and MBeanStatDescriptor
            // Note that wlmModule is only available in ND version
            StatDescriptor sd = new StatDescriptor(new String[] {"wlmModule.server"});
            MBeanStatDescriptor msd = new MBeanStatDescriptor(wlmOName, sd);
            Stats wlmStat = pmiClnt.getStats(nodeName, serverName, msd, false);
            if (wlmStat != null)
                System.out.println("\n\n WLM server data\n\n + " + wlmStat.toString());
            else
                System.out.println("\n\n No WLM server data is availalbe.");

            // how to find all the MBeanStatDescriptors
            testListStatMembers(serverOName);

            // how to use update method
            testUpdate(jvmOName, false, true);
        }
        else {
            System.out.println("No ObjectNames returned from Query" );
        }

    }
    catch(Exception e) {
        new AdminException(e).printStackTrace();
        System.out.println("Exception = " +e);
        e.printStackTrace();
        success = false;
    }


    if(success)
        System.out.println("\n\n All tests are passed");
    else
        System.out.println("\n\n Some tests are failed. Check for the exceptions");

}

/**
 * construct an array from the ArrayList
 */
private static MBeanStatDescriptor[] getMBeanStatDescriptor(ArrayList msds) {
    if(msds == null || msds.size() == 0)
        return null;

    MBeanStatDescriptor[] ret = new MBeanStatDescriptor[msds.size()];
    for(int i=0; i<ret.length; i++)
        if(msds.get(i) instanceof ObjectName)
            ret[i] = new MBeanStatDescriptor((ObjectName)msds.get(i));
        else
            ret[i] = (MBeanStatDescriptor)msds.get(i);
    return ret;
}

/**
 * Sample code to navigate and display the data value from the Stats object.
 */
private static void processStats(Stats stat) {
    processStats(stat, "");
```

```
    }

    /**
     * Sample code to navigate and display the data value from the Stats object.
     */
    private static void processStats(Stats stat, String indent) {
        if(stat == null)  return;

        System.out.println("\n\n");

        // get name of the Stats
        String name = stat.getName();
        System.out.println(indent + "stats name=" + name);

        // Uncomment the following lines to list all the data names
        /*
        String[] dataNames = stat.getStatisticNames();
        for (int i=0; i<dataNames.length; i++)
            System.out.println(indent + "    " + "data name=" + dataNames[i]);
        System.out.println("\n");
        */

        // list all datas
        com.ibm.websphere.management.statistics.Statistic[] allData = stat.getStatistics();

        // cast it to be PMI's Statistic type so that we can have get more
        Statistic[] dataMembers = (Statistic[])allData;
        if(dataMembers != null) {
            for(int i=0; i<dataMembers.length; i++)  {
                System.out.print(indent + "    " + "data name="
+ PmiClient.getNLSValue(dataMembers[i].getName())
                                 + ", description="
+ PmiClient.getNLSValue(dataMembers[i].getDescription())
                                 + ", unit=" + PmiClient.getNLSValue(dataMembers[i].getUnit())
                                 + ", startTime=" + dataMembers[i].getStartTime()
                                 + ", lastSampleTime=" + dataMembers[i].getLastSampleTime());
                if(dataMembers[i].getDataInfo().getType() == TYPE_LONG) {
                    System.out.println(", count="
+ ((CountStatisticImpl)dataMembers[i]).getCount());
                }
                else if(dataMembers[i].getDataInfo().getType() == TYPE_STAT) {
                    TimeStatisticImpl data = (TimeStatisticImpl)dataMembers[i];
                    System.out.println(", count=" + data.getCount()
                                       + ", total=" + data.getTotal()
                                       + ", mean=" + data.getMean()
                                       + ", min=" + data.getMin()
                                       + ", max=" + data.getMax());
                }
                else if(dataMembers[i].getDataInfo().getType() == TYPE_LOAD) {
                    RangeStatisticImpl data = (RangeStatisticImpl)dataMembers[i];
                    System.out.println(", current=" + data.getCurrent()
                                       + ", lowWaterMark=" + data.getLowWaterMark()
                                       + ", highWaterMark=" + data.getHighWaterMark()
                                       + ", integral=" + data.getIntegral()
                                       + ", avg=" + data.getMean());
                }
            }
        }

        // recursively for sub-stats
        Stats[] substats = (Stats[])stat.getSubStats();
        if(substats == null || substats.length == 0)
            return;
        for(int i=0; i<substats.length; i++) {
            processStats(substats[i], indent + "    ");
        }
    }
```

```
    /**
     * test set level and verify using get level
     */
    private static void testSetLevel(ObjectName mbean) {
        System.out.println("\n\n testSetLevel\n\n");
        try {
            // set instrumentation level to be high for the mbean
            MBeanLevelSpec spec = new MBeanLevelSpec(mbean, null, PmiConstants.LEVEL_HIGH);
            pmiClnt.setStatLevel(nodeName, serverName, spec, true);
            System.out.println("after setInstrumentaionLevel high on server MBean\n\n");

            // get all instrumentation levels
            MBeanLevelSpec[] mlss = pmiClnt.getStatLevel(nodeName, serverName, mbean, true);

            if(mlss == null)
                System.out.println("error: null from getInstrumentationLevel");
            else {
                for(int i=0; i<mlss.length; i++)
                    if(mlss[i] != null) {
                        // get the ObjectName, StatDescriptor,
and level out of MBeanStatDescriptor
                        int mylevel = mlss[i].getLevel();
                        ObjectName myMBean = mlss[i].getObjectName();
                        StatDescriptor mysd = mlss[i].getStatDescriptor();  // may be null
                        // Uncomment it to print all the mlss
                        //System.out.println("mlss " + i + ":, " + mlss[i].toString());
                    }
            }
        }
        catch(Exception ex) {
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
            System.out.println("Exception in testLevel");
            success = false;
        }
    }

    /**
     * Use listStatMembers method
     */
    private static void testListStatMembers(ObjectName mbean) {

        System.out.println("\n\ntestListStatMembers \n");
        // listStatMembers and getStats
        // From server MBean until the bottom layer.
        try {
            MBeanStatDescriptor[] msds = pmiClnt.listStatMembers(nodeName, serverName, mbean);
            if(msds == null) return;
            System.out.println(" listStatMembers for server MBean, num members
(i.e. top level modules) is " + msds.length);


            for(int i=0; i<msds.length; i++) {
                if(msds[i] == null)  continue;

                // get the fields out of MBeanStatDescriptor if you need them
                ObjectName myMBean = msds[i].getObjectName();
                StatDescriptor mysd = msds[i].getStatDescriptor();      // may be null

                // uncomment if you want to print them out
                //System.out.println(msds[i].toString());
            }

            for(int i=0; i<msds.length; i++) {
                if(msds[i] == null)  continue;
                System.out.println("\n\nlistStatMembers for msd=" + msds[i].toString());
```

```
                MBeanStatDescriptor[] msds2 =
pmiClnt.listStatMembers(nodeName, serverName, msds[i]);

                // you get msds2 at the second layer now and the
listStatMembers can be called recursively
                // until it returns now.
            }

        }
        catch(Exception ex) {
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
            System.out.println("Exception in testListStatMembers");
            success = false;
        }

    }

    /**
     * Test getStats method
     */
    private static void testGetStats(ArrayList mbeans) {
        System.out.println("\n\n testgetStats\n\n");
        try {
            Stats[] mystats = pmiClnt.getStats(nodeName,
serverName, getMBeanStatDescriptor(mbeans), true);

            // navigate each of the Stats object and get/display the value
            for(int k=0; k<mystats.length; k++) {
                processStats(mystats[k]);
            }

        }
        catch(Exception ex) {
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
            System.out.println("exception from testGetStats");
            success = false;
        }
    }

    /**
     * Test getStats method
     */
    private static void testGetStats2(ObjectName[] mbeans) {
        System.out.println("\n\n testGetStats2\n\n");
        try {
            Stats[] statsArray = pmiClnt.getStats(nodeName, serverName, mbeans, true);

            // You can call toString to simply display all the data
            if(statsArray != null) {
                for(int k=0; k<statsArray.length; k++)
                    System.out.println(statsArray[k].toString());
            }
            else
                System.out.println("null stat");
        }
        catch(Exception ex) {
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
            System.out.println("exception from testGetStats2");
            success = false;
        }
    }

    /**
     * test update method
```

```
     */
    private static void testUpdate(ObjectName oName, boolean keepOld,
boolean recursiveUpdate) {
        System.out.println("\n\n testUpdate\n\n");
        try {
            // set level to be NONE
            MBeanLevelSpec spec = new MBeanLevelSpec(oName, null, PmiConstants.LEVEL_NONE);
            pmiClnt.setStatLevel(nodeName, serverName, spec, true);


            // get data now - one is non-recursive and the other is recursive
            Stats stats1 = pmiClnt.getStats(nodeName, serverName, oName, false);
            Stats stats2 = pmiClnt.getStats(nodeName, serverName, oName, true);

            // set level to be HIGH
            spec = new MBeanLevelSpec(oName, null, PmiConstants.LEVEL_HIGH);
            pmiClnt.setStatLevel(nodeName, serverName, spec, true);

            Stats stats3 = pmiClnt.getStats(nodeName, serverName, oName, true);
            System.out.println("\n\n stats3 is");
            processStats(stats3);

            stats1.update(stats3, keepOld, recursiveUpdate);
            System.out.println("\n\n update stats1");
            processStats(stats1);

            stats2.update(stats3, keepOld, recursiveUpdate);
            System.out.println("\n\n update stats2");
            processStats(stats2);

        }
        catch(Exception ex) {
            System.out.println("\n\n Exception in testUpdate");
            ex.printStackTrace();
            success = false;
        }

    }


}
```

Develop performance monitoring applications with the PMI servlet

The performance servlet uses the Performance Monitor Infrastructure (PMI) to retrieve the performance information from WebSphere Application Server.

The performance servlet EAR file, perfServletApp.ear, is located in the /QIBM/ProdData/WebAS5/Base/installableApps directory. This servlet is deployed like other servlets. For information on deploying applications, see Install and uninstall applications. To use the performance servlet, follow these steps:

1. Deploy the servlet on a single application server instance within the domain.
2. After you deploy the servlet, you can invoke it to retrieve performance data for the WebSphere Application Server domain. To invoke the performance servlet, access this default URL:

   `http://host:port/wasPerfTool/servlet/perfservlet`

   where *host* is the name or IP address of your iSeries server and *port* is the internal HTTP server port for your instance.

The performance servlet provides performance data output as an XML document, as described by the provided document type definition (DTD). The output structure provided is called leaves. The paths that lead to the leaves provide the context of the data. See “PMI servlet” on page 36 for more information about the PMI servlet output.

PMI servlet

The Performance Monitoring Infrastructure (PMI) servlet is used for simple end-to-end retrieval of performance data that any tool, provided by either IBM or a third-party vendor, can handle.

The PMI servlet provides a way to use an HTTP request to query the performance metrics for an entire WebSphere Application Server administrative domain. Because the servlet provides the performance data through HTTP, issues such as firewalls are trivial to resolve.

The performance servlet provides the performance data output as an XML document, as described in the provided document type description (DTD). In the XML structure, the leaves of the structure provide the actual observations of performance data and the paths to the leaves that provide the context. There are three types of leaves or output formats within the XML structure:

- PerfNumericInfo (page 36)
- PerfStatInfo (page 36)
- PerfLoadInfo (page 37)

PerfNumericInfo

When each invocation of the performance servlet retrieves the performance values from PMI, some of the values are raw counters that record the number of times a specific event occurs during the lifetime of the server. If a performance observation is of the type PerfNumericInfo, the value represents the raw count of the number of times this event has occurred after the server started. This information is important to note because the analysis of a single document of data provided by the performance servlet might not be useful for determining the current load on the system. To determine the load during a specific interval of time, it might be necessary to apply simple statistical formulas to the data in two or more documents provided during this interval. The PerfNumericInfo type has these attributes:

- time—Specifies the time when the observation was collected (Java System.currentTimeMillis)
- uid—Specifies the PMI identifier for the observation
- val—Specifies the raw counter value

The following document fragment represents the number of loaded servlets. The path providing the context of the observation is not shown.

```
<numLoadedServlets>
  <PerfNumericData time="988162913175" uid="pmi1"
      val="132"/>
</numLoadedServlets>
```

PerfStatInfo

When each invocation of the performance servlet retrieves the performance values from PMI, some of the values are stored as statistical data. Statistical data records the number of occurrences of a specific event, as the PerfNumericInfo type does. In addition, this type has sum of squares, mean, and total for each observation. This value is relative to when the server started. The PerfStatInfo type has these attributes:

- time—Specifies the time the observation was collected (Java System.currentTimeMillis)
- uid—Specifies the PMI identifier for this observation
- num—Specifies the number of observations
- sum_of_squares—Specifies the sum of the squares of the observations
- total—Specifies the sum of the observations
- mean—Specifies the mean (total number) for this counter

The following fragment represents the response time of an object. The path providing the context of the observation is not shown:

```
<responseTime>
     <PerfStatInfo mean="1211.5" num="5"
sum_of_squares="3256265.0"
time="9917644193057" total="2423.0"
uid="pmi13"/>
</responseTime>
```

PerfLoadInfo

When each invocation of the performance servlet retrieves the performance values from PMI, some of the values are stored as a load. Loads record values as a function of time; they are averages. This value is relative to when the server started. The PerfLoadInfo type has these attributes:

- time—Specifies the time when the observation was collected (Java System.currentTimeMillis)
- uid—Specifies the PMI identifier for this observation
- currentValue—Specifies the current value for this counter
- integral—Specifies the time-weighted sum
- timeSinceCreate—Specifies the elapsed time in milliseconds after this data was created in the server
- mean—Specifies time-weighted mean (integral/timeSinceCreate) for this counter

The following fragment represents the number of concurrent requests. The path providing the context of the observation is not shown:

```
<poolSize>
  <PerfLoadInfo currentValue="1.0" integral="534899.0"
  mean="0.9985028962051592" time="991764193057"
  timeSinceCreate="535701.0 "uid="pmi5"/>
</poolSize>
```

Collect PMI data with the PMI servlet

When the performance servlet is first initialized, it retrieves the list of nodes and servers located within the domain in which it is deployed. The collection of this data is resource intensive. As a result, the performance servlet retains this information as a cached list. If you add a new node to the domain or start a new server, the performance servlet does not automatically retrieve the information about the newly created element. To force the servlet to refresh its configuration, you must add the refreshConfig parameter to the invocation:

```
http://host:port/wasPerfTool/servlet/perfservlet?refreshConfig=true
```

By default, the performance servlet collects all of the performance data for a WebSphere Application Server domain. However, it is possible to limit the data returned by the servlet to a specific node, server, or PMI module.

Node

The servlet can use the node parameter to limit the information it provides to a specific host. For example, to limit the data collection to the node rjones, invoke this URL:

```
http://host:port/wasPerfTool/servlet/perfservlet?Node=rjones
```

Server

The servlet can use the server parameter to limit the information it provides to a specific server. For example, to limit the data collection to the TradeApp server on all nodes, invoke this URL:

```
http://host:port/wasPerfTool/servlet/perfservlet?Server=TradeApp
```

To limit the data collection to the TradeApp server located on the host rjones, invoke this URL:

```
http://host:port/wasPerfTool/servlet/perfservlet?Node=rjones&Server=TradeApp
```

Module

The servlet can use the module parameter limit the information it provides to a specific PMI module. You can request multiple modules from the following Web site:

```
http://host:port/wasPerfTool/servlet/perfservlet?Module=beanModule+jvmRuntimeModule
```

For example, to limit the data collection to the beanModule on all servers and nodes, invoke this URL:

```
http://host:port/wasPerfTool/servlet/perfservlet?Module=beanModule
```

To limit the data collection to the beanModule on the server TradeApp on the node rjones, invoke this URL:

```
http://host:port/wasPerfTool/servlet/perfservlet?Node=rjones&Server=TradeApp
  &Module=beanModule>
```

Access PMI data with the JMX interface

You can use the AdminClient Java Management Extension (JMX) interface to invoke methods on MBeans. The AdminClient API can retrieve Performance Monitoring Infrastructure (PMI) data with PerfMBean or with individual MBeans.

Individual MBeans provide the Stats attribute from which you can get PMI data. PerfMBean provides extended methods for PMI administration and more efficient ways to access PMI data. To set the PMI module instrumentation level, you must invoke methods on PerfMBean. To query PMI data from multiple MBeans, it is faster to invoke the getStatsArray method in PerfMBean than to get the Stats attribute from multiple individual MBeans. PerfMBean can deliver PMI data in a single JMX cell, but multiple JMX calls must be made through individual MBeans.

If you want to retrieve PMI data with an individual MBean, see Use individual MBeans (page 40).

Note: Read the "Code example disclaimer" on page 81 for important legal information.

Use PerfMBean

After the performance monitoring service is enabled and the application server is started or restarted, a PerfMBean is located in each application server. This PerfMBean provides access to PMI data. To use PerfMBean, follow these steps:

1. **Create an instance of AdminClient.**
   When you use the AdminClient API, you must pass the host name, port number and connector type. The example code is:
   ```
   AdminClient ac = null;
     java.util.Properties props = new java.util.Properties();
     props.put(AdminClient.CONNECTOR_TYPE, connector);
     props.put(AdminClient.CONNECTOR_HOST, host);
     props.put(AdminClient.CONNECTOR_PORT, port);
     try {
         ac = AdminClientFactory.createAdminClient(props);
     }
     catch(Exception ex) {
         failed = true;
         new AdminException(ex).printStackTrace();
         System.out.println("getAdminClient: exception");
     }
   ```
2. **Use AdminClient to query the MBean ObjectNames.**
   After you get the AdminClient instance, you can call queryNames to get a list of MBean ObjectNames based on a query string. To get all ObjectNames, you can use the following example code. If you specify a query string, you get a subset of ObjectNames.

```
javax.management.ObjectName on = new javax.management.ObjectName("WebSphere:*");
Set objectNameSet= ac.queryNames(on, null);
// you can check properties like type, name, and process to find a specified ObjectName
```

After you get all the ObjectNames, you can use this example code to get all of the node names:

```
HashSet nodeSet = new HashSet();
  for(Iterator i = objectNameSet.iterator(); i.hasNext(); on =
(ObjectName)i.next()) {
    String type = on.getKeyProperty("type");
    if(type != null && type.equals("Server")) {
    nodeSet.add(servers[i].getKeyProperty("node"));
    }
  }
```

Note: This step returns only the nodes that are running.

To list running servers on the node, you can either check the node name and type for all ObjectNames or use the following example code:

```
StringBuffer oNameQuery= new StringBuffer(41);
    oNameQuery.append("WebSphere:*");
    oNameQuery.append(",type=").append("Server");
    oNameQuery.append(",node=").append(mynode);

    oSet= ac.queryNames(new ObjectName(oNameQuery.toString()), null);
    Iterator i = objectNameSet.iterator ();
      while (i.hasNext ()) {
      on=(objectName) i.next();
    String process= on[i].getKeyProperty("process");
    serversArrayList.add(process);
    }
```

3. **Get the PerfMBean ObjectName for the application server from which you want to get PMI data.**
   Use this example code:

```
for(Iterator i = objectNameSet.iterator(); i.hasNext(); on = (ObjectName)i.next()) {
// First make sure the node name and server name is what you want
// Second, check if the type is Perf
  String type = on.getKeyProperty("type");
  String node = on.getKeyProperty("node");
  String process= on.getKeyProperty("process");
  if (type.equals("Perf") && node.equals(mynode) & & server.equals(myserver)) {
    perfOName = on;
    }
  }
```

4. **Invoke operations on PerfMBean through the AdminClient.**
   After you get one or more PerfMBeans in the application server from which you want to get PMI data, you can use the AdminClient API to invoke these operations on PerfMBean:

```
- setInstrumentationLevel: set the instrmentation level
    params[0] = new MBeanLevelSpec(objectName, optionalSD, level);
    params[1] = new Boolean(true);
    signature= new String[]{ "com.ibm.websphere.pmi.stat.MBeanLevelSpec",
      "java.lang.Boolean"};
    ac.invoke(perfOName, "setInstrumentationLevel", params, signature);

- getInstrumentationLevel: get the instrumentation level
    Object[] params = new Object[2];
    params[0] = new MBeanStatDescriptor(objectName, optionalSD);
    params[1] = new Boolean(recursive);
    String[] signature= new String[]{
      "com.ibm.websphere.pmi.stat.MBeanStatDescriptor", "java.lang.Boolean"};
    MBeanLevelSpec[] mlss = (MBeanLevelSpec[])ac.invoke(perfOName,
      "getInstrumentationLevel", params, signature);

- getConfigs: get PMI static config info for all the MBeans
    configs = (PmiModuleConfig[])ac.invoke(perfOName, "getConfigs", null, null);

- getConfig: get PMI static config info for a specific MBean
```

```
ObjectName[] params = {objectName};
String[] signature= { "javax.management.ObjectName" };
config = (PmiModuleConfig)ac.invoke(perfOName, "getConfig", params,
  signature);

- getStatsObject: you can use either ObjectName or MBeanStatDescriptor
    Object[] params    = new Object[2];
    params[0] = objectName;  // either ObjectName or or MBeanStatDescriptor
    params[1] = new Boolean(recursive);
    String[] signature = new String[] { "javax.management.ObjectName",
      "java.lang.Boolean"};
    Stats stats  = (Stats)ac.invoke(perfOName, "getStatsObject", params,
      signature);

- getStatsArray: you can use either ObjectName or MBeanStatDescriptor
    ObjectName[] onames = new ObjectName[]{objectName1, objectName2};
    Object[] params = new Object[]{onames, new Boolean(true)};
    String[] signature = new String[]{"[Ljavax.management.ObjectName;",
      "java.lang.Boolean"};
    Stats[] statsArray = (Stats[])ac.invoke(perfOName, "getStatsArray",
      params, signature);

- listStatMembers: navigate the PMI module trees

    Object[] params = new Object[]{mName};
    String[] signature= new String[]{"javax.management.ObjectName"};
    MBeanStatDescriptor[] msds = (MBeanStatDescriptor[])ac.invoke(perfOName,
      "listStatMembers", params, signature);

or,

    Object[] params = new Object[]{mbeanSD};
    String[] signature= new String[]
      {"com.ibm.websphere.pmi.stat.MBeanStatDescriptor"};
    MBeanStatDescriptor[] msds = (MBeanStatDescriptor[])ac.invoke
      (perfOName, "listStatMembers", params, signature);
```

Note: For getStatsObject and getStatsArray, the returned data only have dynamic information (value and time stamp). See PmiJmxTest.java for additional code to link the configuration information with the returned data.

Use individual MBeans

To use an individual MBean, follow these steps:

1. Get the AdminClient instance and the ObjectName for the MBean.
2. Get the Stats attribute on the MBean.

Run monitoring applications

After you develop a performance monitoring application, you can being using it to collect PMI data. If you plan to run your performance monitoring application in a secured environment, you must edit the appropriate properties file. For more information, see “Run monitoring applications with security enabled” on page 41.

1. Obtain the pmi.jar and pmiclient.jar files. The pmi.jar and pmiclient.jar files are required for client applications that use PMI client APIs. The pmi.jar and pmiclient.jar files are distributed with WebSphere Application Server and are also a part of WebSphere Java thin client package. You can get these files from either a WebSphere Application Server installation or WebSphere Java Thin Application Client installation. You also need the other JAR files in WebSphere Java Thin Application Client installation in order to run a PMI application.
2. Compile the PMI application, and specify it in the classpath.
   To compile your PMI application, you must include these JAR files in the classpath:
   - admin.jar

- wsexception.jar
- jmxc.jar
- pmi.jar
- pmiclient.jar

If your monitoring applications use APIs in other packages, also include those packages on the classpath.

3. Run the application with this script:

```
call "~dp0setupCmdLine.bat"

set WAS_CP=%WAS_HOME%\lib\properties
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\pmi.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\pmiclient.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\ras.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\admin.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\wasjmx.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\j2ee.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\soap.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\soap-sec.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\nls.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\wsexception.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\ws-config-common.jar
set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\namingclient.jar

%JAVA_HOME%\bin\java "%CLIENTSOAP%" "%CLIENTSAS%" "-Dws.ext.dirs=%WAS_EXT_DIRS%"
  %DEBUGOPTS% -classpath "%WAS_CP%" com.ibm.websphere.pmi.PmiClientTest host
  port connectorType
```

To run the application, run the setupClient script from Qshell to set up the correct environment. For example, if your program is named PmiClientTest, run these commands:

```
STRQSH
. /QIBM/ProdData/WebAS5/Base/bin/setupClient -instance instance
java $JAVA_FLAGS_EXT -classpath cp PmiClientTest host name [port] [connectorType]
```

where *instance* is the name of you WebSphere Application Server instance and *cp* is a classpath that contains the PmiClientTest program.

Run monitoring applications with security enabled

To run your performance monitoring application in a secured environment, you must specify security information in the appropriate properties file.

Note: Do not leave extra spaces at the ends of the lines in the properties files.

Run monitoring applications from your iSeries server

On the iSeries server, the soap.client.props and sas.client.props files are located in the /QIBM/UserData/WebAS5/*edition*/*instance*/properties directory, where *edition* is Base for WebSphere Application Server and ND for WebSphere Application Server Network Deployment, and *instance* is the name of your WebSphere Application Server instance.

- If you use SOAP to access the application server, make these changes to the soap.client.props file:
  1. Set the com.ibm.SOAP.securityEnabled property to `true`.
  2. Set com.ibm.SOAP.loginUserid and com.ibm.SOAP.loginPassword properties to a valid user ID and password, respectively.
- If you use RMI to access the application server, make these changes to the sas.client.props file:
  1. Set the com.ibm.CORBA.securityEnabled property to `true`.
  2. Set the com.ibm.CORBA.loginSource property to `properties`.
  3. Set the com.ibm.CORBA.loginUserid and com.ibm.CORBA.loginPassword properties to a valid user ID and password, respectively.

Note: These properties can also be passed as Java properties on the Java command line call to your program.

Run monitoring applications from a workstation

If you are running a monitoring application from a workstation, you must specify security information in the properties files on the workstation. The soap.client.props and sas.client.props files are located on your workstation in the *was_install_root*/properties directory, where *was_install_root* is the directory where you installed the workstation components of WebSphere Application Server.

- For Tivoli Performance Viewer, make these changes to the soap.client.props file:
  1. Set the com.ibm.SOAP.securityEnabled property to `true`.
  2. Set com.ibm.SOAP.loginUserid and com.ibm.SOAP.loginPassword properties to a valid user ID and password, respectively.
- For other performance monitoring applications, or if you are using RMI to access the application server, make these changes to the sas.client.props file:
  1. Set the com.ibm.CORBA.securityEnabled property to `true`.
  2. Set the com.ibm.CORBA.loginSource property to `properties`.
  3. Set the com.ibm.CORBA.loginUserid and com.ibm.CORBA.loginPassword properties to a valid user ID and password, respectively.

  Note: These properties can also be passed as Java properties on the Java command line call to your program.

Primary tuning parameters

It is recommended that you review these primary tuning parameters before you check other parameters. These parameters can have a significant impact on performance. Because they are specific to an application, the appropriate parameter settings for different applications and environments can vary.

- "Hardware capacity and configuration" on page 62
- "Web server tuning parameters" on page 70
- "Database tuning parameters" on page 71
- Java virtual machine heap size (page 63)
- "Application assembly performance checklist" on page 72
- Pass by value/Pass by reference (page 44)
- ORB thread pool size (page 44)
- Web container thread pool size (page 45)
- HTTP keep alive connections
- Data sources connection pool and prepared statement cache (page 48)

Tuning parameters for your WebSphere Application Server environment

Each application server instance has several parameters that can influence application performance. You can use the WebSphere Application Server administrative console to configure and tune applications, Web containers, EJB containers, application servers, and nodes in the administrative domain. You can also configure settings for other components of your iSeries environment to optimize performance.

If you experience problems with performance, it is recommended that you review the "Primary tuning parameters" before you review this index.

- WebSphere Application Server tuning parameters (page 43)
- Additional tuning parameters (page 43)

WebSphere Application Server tuning parameters

Each application server instance has several parameters that can influence application performance. You can use the WebSphere Application Server administrative console to configure and tune applications, Web containers, EJB containers, application servers, and nodes in the administrative domain. You can also configure settings for other components of your iSeries environment to optimize performance. These topics describe tuning parameters that are specific to WebSphere Application Server and application server instances.

"Application server tuning parameters"
You can tune parameters for several application server components, such as the Web container and the Object Request Broker. This topic provides information about tuning these parameters.

"Java Messaging Service tuning parameters" on page 50
This topic describes how you can optimize JMS performance.

"Queuing network" on page 54
This topic describes how to tune the components of the queueing network to optimize performance.

"Web server plug-in tuning tips" on page 59
Application server instances use plug-ins to communicate with Web servers. This topic provides tips to help you optimize the performance of the Web server plug-in.

"Web services tuning tips" on page 60
This topic describes considerations for tuning Web services.

Enabling security decreases performance. You can tune your security configuration to minimize this impact. For more information, see Tune your security configuration in *Security*.

Additional tuning parameters

These topics provide information about other components of your environment that can affect WebSphere Application Server performance:

"Hardware capacity and configuration" on page 62
This topic provides information about configuring your hardware for optimal performance.

"Java virtual machine tuning parameters" on page 63
This topic describes tuning parameters for the OS/400 Java virtual machine and other tips for optimizing Java application performance.

"Web server tuning parameters" on page 70
This topic provides information about optimizing the performance of your Web server.

"Database tuning parameters" on page 71
This topic provides information about tuning database performance.

"TCP/IP buffer sizes" on page 71
This topic describes how TCP/IP buffer sizes can affect performance.

Application server tuning parameters

Each application server instance has several parameters that can influence application performance. You can use the WebSphere Application Server administrative console to configure and tune applications, Web containers, EJB containers, application servers, and nodes in the administrative domain.

You can tune application server settings to control how an application server provides services for running applications and their components. Each application server instance contains interrelated components, called a queuing network, that must be properly tuned to support the specific needs of your e-business application. The queuing network helps the system achieve maximum throughput and maintain the overall stability of the system. For more information about the queuing network, see "Queuing network" on page 54.

You can tune the following application server settings:
- Object Request Broker (page 44)
- Dynamic cache service (page 45)
- Web container (page 45)
- EJB container (page 47)
- Session management (page 48)
- Data sources (page 48)
- Process Priority (page 49)

Object Request Broker

An Object Request Broker (ORB) uses the Internet InterORB Protocol (IIOP) to manage the interaction between clients and servers. It supports client requests and responses received from servers in a network-distributed environment. Object Request Broker tuning guidelines provides tips on using these parameters to tune the ORB. You can tune the following ORB parameters:
- **Pass-by-reference (com.ibm.CORBA.iiop.noLocalCopies)**
  - Description: For remote method calls, this parameter specifies that the ORB passes parameters by reference instead of by value. If pass-by-reference is disabled, the ORB copies parameters to the stack before every remote method call is made. For more information, see Object Request Broker service settings in administrative console.
  - How to view and set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **ORB Service**.
    5. On the **ORB Service** page, select the **Pass by reference** check box.
    6. Click **Apply** or **OK**.
    7. Save the configuration.
    8. Stop and restart the application server.
  - Default value: Disabled
  - Recommended value: Enabling pass-by-reference can enhance performance, but is not specification compliant. Some applications may not run correctly if you enable pass-by-reference. For more information, see Restrictions for using the pass-by-reference option. Enterprise bean local interfaces can provide many of the performance benefits of pass-by-reference in a specification compliant manner. For more information about local interfaces, see Container-managed persistence features.
- **Connection cache maximum (com.ibm.CORBA.MaxOpenConnections)**
  - Description: This parameter specifies the largest number of connections allowed to occupy the service's connection cache.
  - How to view and set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.

4. Click **ORB Service**.
5. On the **ORB Service** page, specify a value in the **Connection cache maximum** field.
6. Click **Apply** or **OK**.
7. Save the configuration.
8. Stop and restart the application server.

– Default value: 240
– Recommended value: Adjust this value as necessary to support the number of clients that connect to the server-side ORB. You can increase this value to support up to 1000 clients.

- **Thread pool Maximum size**
  – Description: This value specifies the maximum number of threads in the pool. Each call to an EJB method from a separate JVM (either on the same system or another system) uses an ORB thread. If your application calls EJB methods only from servlets in the same JVM, you probably do not need to tune this parameter. For more information, see Thread pool settings
  – How to view and set:
  1. Start the administrative console.
  2. In the topology tree, expand **Servers** and click **Application Servers**.
  3. Click the name of the application server that you want to configure.
  4. Click **ORB Service**.
  5. On the **ORB Service** page, click **Thread Pool**.
  6. Specify a value for the **Maximum size** field.
  7. Click **Apply** or **OK**.
  8. Save the configuration.
  9. Stop and restart the application server.

  – Default value: 50
  – Recommended value: Increase this value if the Percent Maxed metric in Tivoli Performance Viewer is consistently greater than 10.
- **com.ibm.CORBA.ServerSocketQueueDepth** This is a custom property for the ORB Service. For more information, see
- **com.ibm.CORBA.FragmentSize** This is a custom property for the ORB Service. For more information, see Set custom properties for the ORB service.

You can also tune several custom paramters for the ORB service. For more information, see Set custom properties for the ORB service.

Dynamic cache service

Dynamic caching maintains server-generated content in memory for servlets, JavaServer Pages (JSP) files, commands, and web services. Items that are called frequently with identical parameters are good candidates for dynamic caching. Dynamic caching is ideal for Web sites that use mass personalization or have a very high number of calls to a particular set of servlets or JSP files. Because the application does not have to generate the output every time a servlet or JSP file is called, the dynamic cache service can improve performance. For information about enabling and configuring the dynamic cache service see Dynamic cache service in *Application development*.

Web container

Each application server includes a Web container. The application server routes servlet requests along a transport queue between the Web server plug-in and the Web container. Default Web container properties

are set for simple Web applications. However, these values might not be appropriate for more complex Web applications. You can adjust these parameters to tune the Web container based on the specific needs of your environment:

- **Thread pool Maximum size**
  - Description: This value limits the number of requests that your application server can process concurrently. For more information, see Thread pool settings.

    Note: The information on the settings page applies to all thread pools in WebSphere Application Server.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Web Container**.
    5. On the **Web Container** page, click **Thread Pool**.
    6. Specify a value for the **Maximum size** field.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: 50
  - Recommended value: This value should be set to handle the peak load on your application server. It is recommended that you specify a maximum size less than or equal to the number of threads processing requests in your HTTP server. A value in the range 25-50 is generally a good starting point. You can use the Tivoli Performance Viewer to monitor the number of threads being used.
- **Growable thread pool**
  - Description: This setting specifies whether the number of threads can increase beyond the maximum size configured for the thread pool. For more information, see Thread pool settings.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Web Container**.
    5. On the **Web Container** page, click **Thread Pool**.
    6. Select **Allow thread allocation beyond maximum thread size**.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: Disabled
  - Recommended value: It is recommended that you do not enable this property if you are confident the Thread pool Maximum size is large enough to adequately process the peak load on your application server. Set to true if you want the thread pool to be able to exceed the configured maximum pool size. This setting is beneficial in the event that the application server receives an unexpected increase in requests or the maximum pool size is set too low. In this scenario, additional threads are created to handle the increased number of requests. These connections are destroyed when the number of requests returns to its typical level. However, enabling the growable thread pool setting might cause a large number of threads to be created, and have a negative impact on system storage and performance.

You can also tune several custom parameters for HTTP transports in the Web container. For more information, see Set custom properties for an HTTP transport.

EJB container

Each application server includes an EJB container. You can use the following parameters to make adjustments that improve performance:

- **Cleanup interval**
  - Description: This parameter specifies the interval at which the container attempts to remove unused items from the cache. The cache manager tries to maintain some unallocated entries that can be allocated as needed. A background thread attempts to free some entries while maintaining some unallocated entries. If the thread runs while the application server is idle, the application server does not need to remove entries from the cache before it can allocate new cache entries. For more information, see EJB cache settings.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **EJB Container**.
    5. On the **EJB Container** page, click **EJB Cache Settings**.
    6. Specify a value for the **Cleanup interval** field.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: 3000
  - Recommended value: It is recommended that you increase the value of parameter as the cache size increases.
- **Cache size**
  - Description: This parameter specifies the number of buckets in the cache. A bucket can contain more than one active enterprise bean instance, but performance is maximized if each bucket in the table has a minimum number of instances assigned to it. When the number of active instances within the container exceeds the cache size, the container periodically attempts to passivate some of the active instances. For more information, see EJB cache settings.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **EJB Container**.
    5. On the **EJB Container** page, click **EJB Cache Settings**.
    6. Specify a value for the **Cache Size** field.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: 2053
  - Recommended value: For the best balance of performance and memory, set this value to the maximum number of active instances expected during a typical workload. To estimate the number of active instances, multiply the number of entity beans active in any given transaction by the total

number of concurrent transactions expected. Then add the number of active session bean instances. You can user the Tivoli Performance Viewer to monitor this setting.

For more information about tuning enterprise bean performance, see these topics:

- Assembling EJB modules
During application assembly, you can break Container Managed Persistence enterprise beans into several enterprise bean modules.
- "Enterprise bean method invocation queuing" on page 58
- Access intent policies

Session management

The installed default settings for session management are configured for optimal performance. For more information, see Tune session management in *Application development* and Tuning parameter settings.

Data sources

Applications uses data sources to access databases. The following data source settings can affect performance:

- **Connection pooling** For moreinformation about connection pooling, see Connection pooling in *Application Development* and Connection pool settings.
  – Maximum connection pool
    - Description: This value specifies the maximum number of managed connections for a pool.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Resources** and click **JDBC Providers**.
      3. Click the name of the provider for the data source that you want to configure.
      4. Click **Data Sources**.
      5. Click the name of the data source that you want to configure.
      6. Click **Connection Pool**.
      7. Specify a value in the **Max Connections** field.
      8. Click **Apply** or **OK**.
      9. Save the configuration.
      10. Stop and restart the application server.
    - Default value: 10
    - Recommended value: Set the value for the connection pool lower than the value for the **Max Connections** option in the Web container (page 45). If the pool is larger than necessary, it might waste memory and other system resources. A setting of 10-25 is suitable for many applications. For additional about connection pool size, see "Queuing network" on page 54. Use Tivoli Performance Viewer monitor connections. When the application reaches a stable state, look for connections that are being closed instead of returned to the pool. For information about monitoring connection pools, see "Monitor performance with Tivoli Performance Viewer" on page 13.
  – Minimum connection pool
    - Description: This value specifies the minimum number of managed connections for a pool.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Resources** and click **JDBC Providers**.

        3. Click the name of the provider for the data source that you want to configure.
        4. Click **Data Sources**.
        5. Click the name of the data source that you want to configure.
        6. Click **Connection Pool**.
        7. Specify a value in the **Min Connections** field.
        8. Click **Apply** or **OK**.
        9. Save the configuration.
        10. Stop and restart the application server.
    - Default value: 1
    - Recommended value: Set the minimum pool size to handle the average load on the system. You can use Tivoli Performance Viewer to monitor the pool.
- **Statement cache size**
  - Description: WebSphere Application Server provides a statement cache for each data source. This value represents the number of free prepared statements per connection in the connection pool. The statement cache stores query information for the data source. You can adjust to size of the statement cache to optimize performance. For more information, see Data source settings.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Resources** and click **JDBC Providers**.
    3. Click the name of the provider for the data source that you want to configure.
    4. Click **Data Sources**.
    5. Click the name of the data source that you want to configure.
    6. Specify a value in the **Statement Cache Size** field.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: 10
  - Recommended value: In most situations, it is recommended that you set this to the number of unique statements for each application that uses the datasource. Setting the cache size to this value avoids cache discards, and generally results in the best performance. However, if the cache is too large, it might cause performance problems as a result of increased cache management and increased use of system resources. If you have a large number of unique statements, a smaller number might be appropriate. Use Tivoli Performance Viewer to monitor the cache size. For information about tuning the statement cache size, see Tuning the WebSphere Prepared Statement Cache.

Process Priority

- Description: The priority setting establishes the Application Server job’s run priority. The default process priority is 25. The application server does not override the default behavior of Java thread creation. Worker threads within the server are configured to run at 6 levels lower than the job’s run priority. Therefore, by default, the priority of the worker threads is 31. For more information about this setting, see Process execution settings.
- How to view or set:
  1. Start the administrative console.
  2. In the topology tree, expand **Servers** and click **Application servers**.
  3. Click the name of the application server that you want to configure.
  4. Click **Process Definition**.

5. Click **Process Execution**.
6. On the **Process Execution** page, specify a value in the **Process Priority** field.
7. Click **Apply** or **OK**.
8. Save the configuration.
9. Stop and restart the application server.

- Default value: 25
- Recommended value: In most situations, the default value is acceptable. However, if other workloads are running at a higher priority (that is, with a lower priority number), you might need to adjust the application server's priority so that it can more easily access the necessary resources.

Java Messaging Service tuning parameters

You can use the administrative console to tune Java Messaging Service (JMS) run-time components, resources, and the embedded messaging server. For more information, see Administer the embedded JMS server and Administer JMS servers in Network Deployment.

Note: If you are not using JMS in your application server, it is recommended that you disable it. (By default, JMS is not enabled in the default instance. See Default instances for more information.) You can disable JMS when you create an application server instance with the crtwasinst script, or with the chgwassvr script in an existing instance. For more information, see The crtwasinst script and The chgwassvr script. The JMS service consumes a low amount of resources during runtime, but can cause significant increases in the start and stop times for your application server.

- **Listener service**
  - Thread pool
    - Description: The thread pool determines the maximum number of threads that the Message Listener Service can run. Adjust this parameter when multiple message-driven beans are deployed in the same application server and the sum of their maximum session values exceeds the default value of 10.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Servers** and click **Application Servers**.
      3. Click the name of the application server that you want to configure.
      4. Click **Message Listener Service**.
      5. On the **Message Listener Service** page, click **Thread Pool**.
      6. Edit the thread pool settings as needed.
      7. Click **Apply** or **OK**.
      8. Save the configuration.
      9. Stop and restart the application server.
    - Default value: Minimum: 10; Maximum: 50
    - Recommended value: Set the minimum to the sum of all message-driven beans maximum session values. Set the maximum to a value equal to or greater than the minimum.
  - Custom properties
    - Application Server Facilities and Non-Application Server Facilities
      - Description: The JMS server has two modes of operation: Application Server Facilities (ASF) and non-ASF. ASF mode is meant to provide concurrency and transactional support for applications. Non-ASF bypasses that support to streamline the path length. The value is the number of milliseconds that are required for a message to be delivered.
      - How to view or set:
        1. Start the administrative console.

2. In the topology tree, expand **Servers** and click **Application Servers**.
3. Click the name of the application server that you want to configure.
4. Click **Message Listener Service**.
5. On the **Message Listener Service** page, click **Thread Pool**.
6. Edit the thread pool settings as needed.
7. Click **Apply** or **OK**.
8. Save the configuration.
9. Stop and restart the application server.

- Default value: ASF mode (no custom property)
- Recommended value: It is recommended that you set this value lower than the transaction timeout by 10 or more seconds. The difference should be larger under extreme loads in which threads are waiting long periods of time to get CPU cycles. Use Non-ASF mode in these situations:
  - Message order is a strict requirement
  - You want concurrent PTP messages
  - The property non.asf.receive.timeout exists and has a value greater than 0

  Do not use Non-ASF mode if you want concurrent publications and subscriptions messages. In this situation, ASF mode provides better throughput.

- **Listener port**
  - Maximum sessions
    - Description: This parameter specifies the maximum number of concurrent JMS server sessions that a listener uses to process messages.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Servers** and click **Application Servers**.
      3. Click the name of the application server that you want to configure.
      4. Click **Message Listener Service**.
      5. On the **Message Listener Service** page, click **Listener Ports**.
      6. Click the name of the listener port.
      7. Specify a value in the **Maximum sessions** field.
      8. Click **Apply** or **OK**.
      9. Save the configuration.
      10. Stop and restart the application server.
    - Default value: 1
    - Recommended value: If you want to use message concurrency (multiple messages processed simultaneously), set the value to 2-4 sessions per system processor. Specify the lowest possible value to eliminate client thrashing. If you want a strict message order, set the value to 4. This value ensures that there is always a thread waiting in a hot state, blocked on receiving the message.
  - Maximum messages
    - Description: This parameter specifies the maximum number of messages that the listener can process in one JMS server session.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Servers** and click **Application Servers**.
      3. Click the name of the application server that you want to configure.

        4. Click **Message Listener Service**.
        5. On the **Message Listener Service** page, click **Listener Ports**.
        6. Click the name of the listener port.
        7. Specify a value in the **Maximum messages** field.
        8. Click **Apply** or **OK**.
        9. Save the configuration.
        10. Stop and restart the application server.
    - Default value: 1
    - Recommended value: If you want to use message concurrency, set the value to 2-4 sessions per system processor. Specify the lowest possible value to eliminate client thrashing. If you want a strict message order, set the value to 1.

- **JMS resources**
  - XA enabled
    - Description: This attribute specifies whether the connection factory is for XA or non-XA coordination of messages. If you set this property to NON_XA, the JMS session is still enlisted in a transaction, but uses the resource manager local transaction calls (session.commit and session.rollback) instead of XA calls. As a result, performance might improve. However, only a single resource can be enlisted in a transaction in WebSphere Application Server.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Resources** and click **WebSphere JMS Provider** or **WebSphere MQ JMS Provider**. This parameter does not apply to generic JMS providers.
      3. Click one of these links, depending on which provider you are using and which type of connection factory you want to configure:
         - **WebSphere Queue Connection Factories**
         - **WebSphere Topic Connection Factories**
         - **WebSphere MQ Queue Connection Factories**
         - **WebSphere MQ Topic Connection Factories**
      4. Click the name of the connection factory that you want to configure.
      5. Select or clear **XA Enabled**.
      6. Click **Apply** or **OK**.
      7. Save the configuration.
      8. Stop and restart the application server.
    - Default value: By default, XA is enabled.
    - Recommended value: Do not enable XA when the message queue or topic received is the only resource in the transaction. Enable XA when other resources, including other queues or topics, are involved.
  - Connection pool size
    - Description: The connection pool size specifies the maximum number of connections that the pool can contain. This parameter does not apply to generic JMS providers.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Resources**.
      3. Navigate through one of these paths to the connection factory that you want to configure:
         - **WebSphere JMS Provider —> WebSphere Queue Connection Factories —>** *conn_factory_name* **—> Connection Pool**

- **WebSphere JMS Provider —> WebSphere Topic Connection Factories —>** *conn_ factory_name* **—> Connection Pool**
- **WebSphere MQ JMS Provider —> WebSphere MQ Queue Connection Factories —>** *conn_ factory_name* **—> Connection Pool**
- **WebSphere MQ JMS Provider —> WebSphere MQ Topic Connection Factories —>** *conn_ factory_name* **—> Connection Pool**

4. Specify a value in the **Max Connections** field.
5. Click **Apply** or **OK**.
6. Save the configuration.
7. Stop and restart the application server.

- Default value: 10
- Recommended value: For the connection pool size, specify a value lower than the value for the Max Connections option in the Web container. Lower settings, such as 10-30 connections, perform better than higher settings, such as 100.

– WebSphere MQ queue connection factory transport type

- Description: The transport type specifies whether the WebSphere MQ client connection or JNI bindings are used for connection to the WebSphere MQ queue manager.
- How to view or set:

1. Start the administrative console.
2. In the topology tree, expand **Resources** and click **WebSphere MQ JMS Provider**.
3. Click **WebSphere MQ Queue Connection Factories**.
4. Click the name of the connection factory that you want to configure.
5. For **Transport Type**, select a value from the drop-down list.
6. Click **Apply** or **OK**.
7. Save the configuration.
8. Stop and restart the application server.

- Default value: Bindings
- Recommended value: BINDINGS is faster by 30% or more, but it lacks security. It is recommended that if you have security concerns, use BINDINGS.

– WebSphere MQ topic connection factory transport type

- Description: The transport type specifies whether the WebSphere MQ client connection or JNI bindings are used for connection to the WebSphere MQ queue manager.
- How to view or set:

1. Start the administrative console.
2. In the topology tree, expand **Resources** and click **WebSphere MQ JMS Provider**.
3. Click **WebSphere MQ Topic Connection Factories**.
4. Click the name of the connection factory that you want to configure.
5. For **Transport Type**, select a value from the drop-down list.
6. Click **Apply** or **OK**.
7. Save the configuration.
8. Stop and restart the application server.

- Default value: Bindings
- Recommended value: DIRECT is the fastest, and is the recommended setting. However, if you wantto satisfy additional security tasks and the queue manager is local to the JMS client, specify BINDINGS. For all other cases, specify QUEUED.

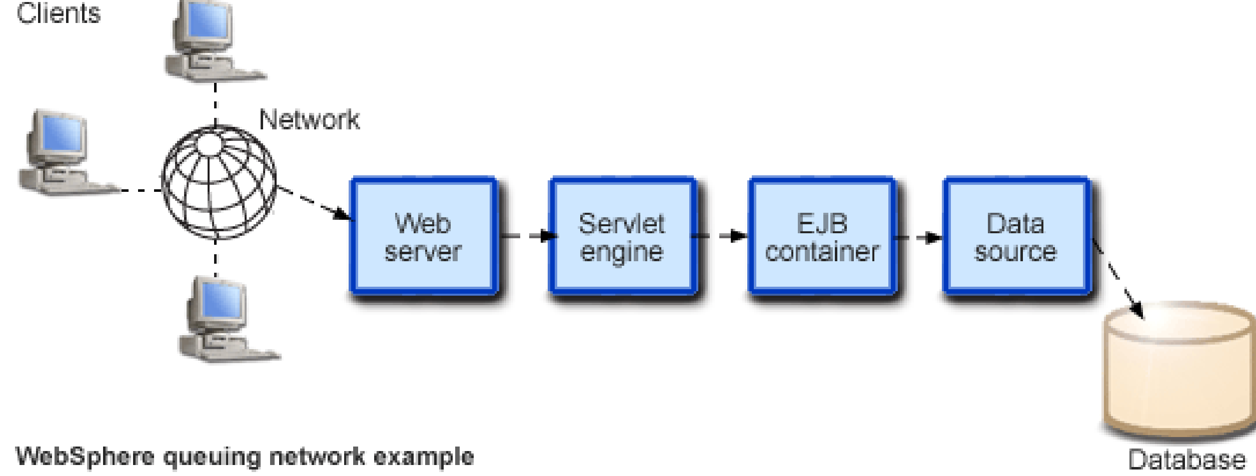
- Transaction log directory
  - Description: This parameter specifies the directory that contains transaction logs. When an application accesses more then one resource, WebSphere Application Server stores transaction information to properly coordinate and manage the distributed transaction.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the applicaiton server that you want to configure.
    4. Click **Transaction Service**.
    5. Specify a value for the **Transaction log directory** field.
    6. Click **Apply** or **OK**.
    7. Save the configuration.
    8. Stop and restart the application server.
  - Default value: /QIBM/UserData/WebAS5/*edition*/*instance*/tranlog, where *edition* is Base for WebSphere Application Server and ND for WebSphere Application Server Network Deployment, and *instance* is the name of your application server instance.
  - Recommended value: Storing transaction data can have a negative impact on performance. To improve performance, you can move the transaction log to a storage device with more physical disk drives or with Redundant Array of Independent Disks (RAID) disk drives.

- **Embedded JMS server**
  - Number of threads
    - Description: With the embedded JMS publications and subscriptions server, this value is the number of threads to use for the publications and subscriptions matching engine, which matches publications to subscribers. Use this parameter when concurrent publications and subscriptions exist that would exceed the capacity of the default value.
    - How to view or set:
      1. Start the administrative console.
      2. In the topology tree, expand **Servers** and click **Application Servers**.
      3. Click the name of the application server that you want to configure.
      4. Click **Server Components**.
      5. Click **JMS Servers**.
      6. Specify a value for the **Number of threads** field.
      7. Click **Apply** or **OK**.
      8. Save the configuration.
      9. Stop and restart the application server.
    - Default value: 1
    - Recommended value: Set this value slightly higher than the number of concurrent message publishers. If large numbers of subscribers exist, increasing this value can also provide some benefit.

Queuing network

WebSphere Application Server contains interrelated components that must be tuned to support the custom needs of your e-business application. These adjustments help the system achieve maximum throughput while maintaining the overall stability of the system. This group of interconnected components is known as a queuing network. These queues or components include the network, Web server, Web container, EJB container, data source, and possibly a connection manager to a custom back-end system. Each of these resources represents a queue of requests waiting to use that resource. Various queue settings include:

- “Web server tuning parameters” on page 70: ThreadsPerChild
- Web container (page 45): Thread pool maximum size, HTTP transports MaxKeepAliveConnections, and MaxKeepAliveRequests
- Object Request Broker (page 44): Thread pool maximum size
- Data source (page 48): Connection pooling and Statement cache size
- “Java Messaging Service tuning parameters” on page 50: Listener service thread pool maximum size, Listener port maximum sessions, Listener port maximum messages, and Connection pooling



Most of the queues that make up the queuing network are closed queues. A closed queue places a limit on the maximum number of requests present in the queue, while an open queue has no limit. A closed queue supports strict management of system resources. For example, the Web container thread pool setting controls the size of the Web container queue. If the average servlet running in a Web container creates 10MB of objects during each request, a value of 100 for thread pools limits the memory consumed by the Web container to 1GB.

In a closed queue, requests can be active or waiting. An active request is doing work or waiting for a response from a downstream queue. For example, an active request in the Web server is doing work, such as retrieving static HTML, or waiting for a request to be processed in the Web container. A waiting request is waiting to become active. The request remains in the waiting state until one of the active requests leaves the queue.

All Web servers supported by WebSphere Application Server are closed queues, as are WebSphere Application Server data sources. You can configure Web containers as open or closed queues. In general, it is recommended that you use closed queues. EJB containers are open queues. If there are no threads available in the pool, a new thread is created for the duration of the request.

If enterprise beans are called by servlets, the Web container limits the number of total concurrent requests into an EJB container, because the Web container also has a limit. The Web container limits the number of total concurrent requests only if enterprise beans are called from the servlet thread of execution. Nothing prevents a servlet from creating threads and overloading the EJB container with requests. Therefore, servlets should not create their own work threads.

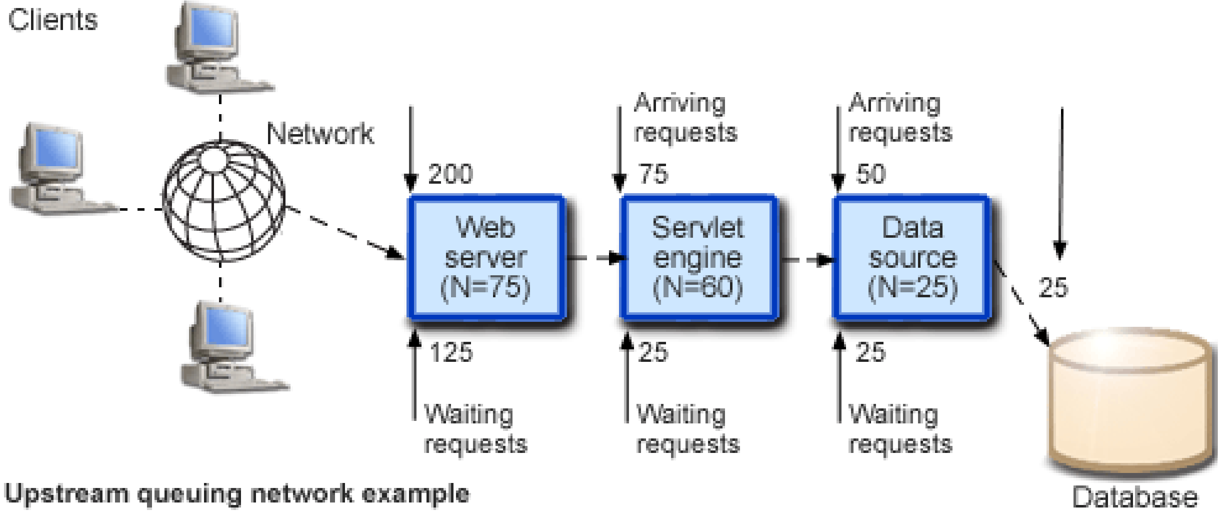
“Queue configuration tips” on page 56
This topic provides tips to help you tune the queuing network configuration.

“Enterprise bean method invocation queuing” on page 58
This topic provides information about invocation queuing for remote method invocations.

Queue configuration tips

This page outlines a methodology for configuring the WebSphere Application Server queues. Moving the database server onto another machine or providing more powerful resources, such as a faster set of CPUs with more memory, can dramatically change the dynamics of your application server environment.

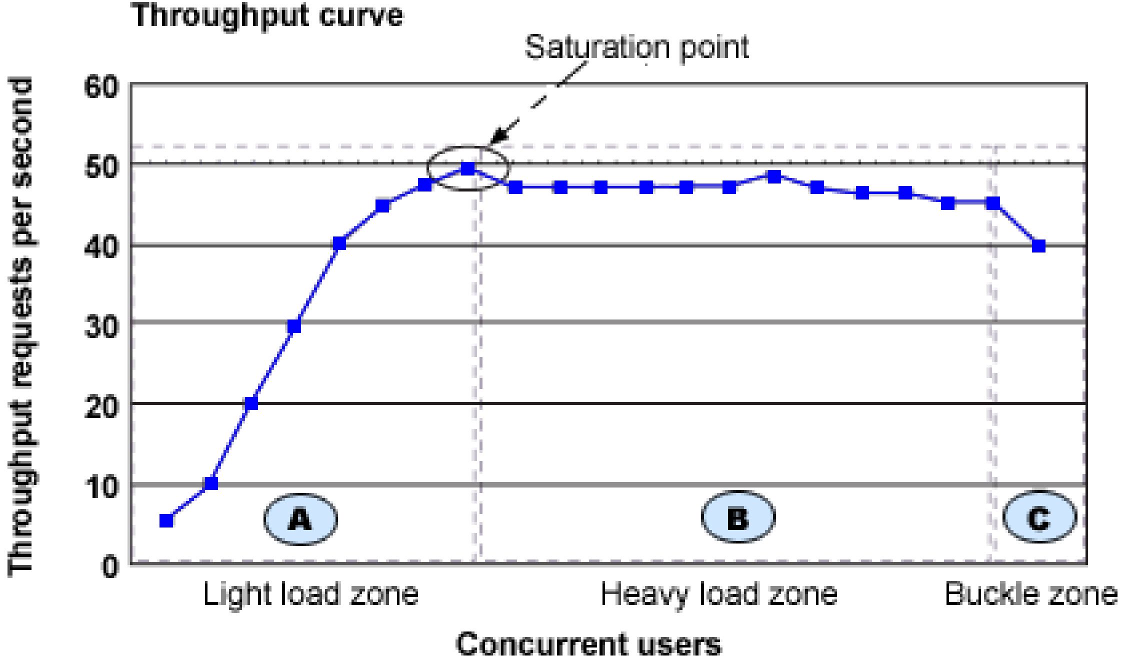
- **Minimize the number of requests in WebSphere Application Server queues.**
  Performance is usually improved if requests wait in the network, ahead of the Web server, rather than waiting in the application server. That is, only requests that can be processed enter the queuing network. To achieve this result, set the size of upstream (closest to the client) queues large, and specify progressively smaller sizes for downstream (further from the client) queues. The figure provides an example of this configuration.



Upstream queuing network example

  Queues in the queuing network become progressively smaller as work flows downstream. In this example, 200 client requests arrive at the Web server. 125 requests remain queued in the network because the Web server is set to handle 75 concurrent clients. As the 75 requests pass from the Web server to the Web container, 25 requests remain queued in the Web server and the remaining 50 are handled by the Web container. This process progresses through the data source until 25 user requests arrive at the final destination, the database server. Because there is work waiting to enter a component at each point upstream, no component in this system must wait for work to arrive. Most of the requests wait in the network, outside of WebSphere Application Server. This type of configuration adds stability, because no component is overloaded.

- **Draw throughput curves to determine when the system capabilities are maximized.**
  To run a test case that represents the full use of the production application, exercise all meaningful code paths or use the production application. Run a set of tests to determine when the system capabilities are fully stressed or when the network has reached the saturation point. Conduct these tests after most of the bottlenecks are removed from the application. The goal of these tests is to drive CPUs to near 100% utilization. For maximum concurrency through the system, start the initial baseline experiment with large queues. For example, start the first experiment with a queue size of 100 at each of the servers in the queuing network: Web server, Web container, and data source. Begin a series of experiments to plot a throughput curve, increasing the concurrent user load after each experiment. For example, perform experiments with 1, 2, 5, 10, 25, 50, 100, 150 and 200 users. After each test, record the throughput requests per second, and response times in seconds per request. The curve resulting from the baseline experiments resembles the following typical throughput curve:



The WebSphere Application Server throughput is a function of the number of concurrent requests present in the total system. Section A, the light load zone, shows that as the number of concurrent user requests increases, the throughput increases almost linearly with the number of requests. Under light loads, concurrent requests face very little congestion within the WebSphere Application Server system queues. At some point, congestion starts to develop and throughput increases at a much lower rate until it reaches a saturation point that represents the maximum throughput value, as determined by some bottleneck in the WebSphere Application Server system. The most manageable type of bottleneck occurs when the WebSphere Application Server machine CPUs become fully utilized. To resolve this bottleneck, you must add processing power.

In the heavy load zone, Section B, as the concurrent client load increases, throughput remains relatively constant. However, the response time increases proportionally to the user load. That is, if the user load is doubled in the heavy load zone, the response time doubles. At some point, represented by Section C, the buckle zone, one of the system components becomes exhausted. At this point, throughput starts to decrease. For example, the system might enter the buckle zone when the network connections at the Web server exhaust the limits of the network adapter or if the requests exceed operating system limits for file handles.

If the saturation point is reached by driving CPU utilization close to 100%, you can move on to the next step. If the saturation point occurs before system utilization reaches 100%, another bottleneck is probably the cause. For example, the application might be creating Java objects and causing excessive garbage collection bottlenecks in the Java code.

There are two ways to manage application bottlenecks: remove the bottleneck or clone the bottleneck. The best way to manage a bottleneck is to remove it. You can use a Java-based application profiler to examine overall object utilization. For a list of available tools, see “Performance tools” on page 3.

- **Decrease queue sizes as requests move downstream from the client.**
  The number of concurrent users at the throughput saturation point represents the maximum concurrency of the application. For example, if the application saturates WebSphere Application Server at 50 users, using 48 users might produce the best combination of throughput and response time. This value is called the Max Application Concurrency value. Max Application Concurrency becomes the preferred value for adjusting the WebSphere Application Server system queues. Remember, it is desirable for most users to wait in the network; therefore, queue sizes should decrease when moving downstream farther from the client. For example, given a Max Application Concurrency value of 48, start with system queues at the following values: Web server 75, Web container 50, data source 45. Perform a set of additional tests with slightly higher and lower values to find the best settings.

To help determine the number of concurrent users, view the Servlet Engine Thread Pool and Concurrently Active Threads metrics in the Tivoli Performance Viewer.

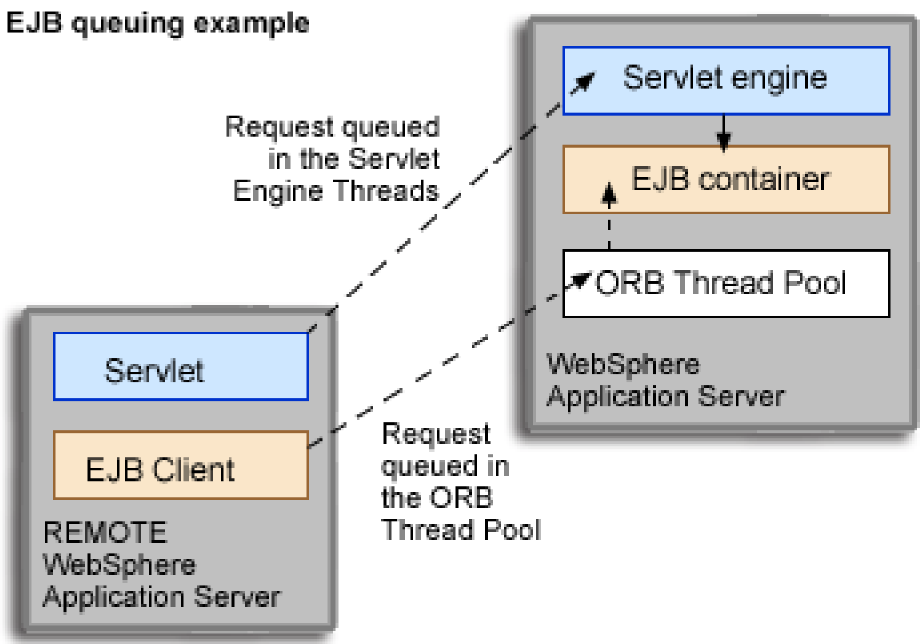
- **Adjust queue settings to correspond to access patterns.**
  In many cases, only a fraction of the requests that pass through one queue enter the next queue downstream. For example, on a Web site with many static pages, a number of requests are fulfilled at the Web server and are not passed to the Web container. In this case, the Web server queue can be significantly larger than the Web container queue. In the previous example, the Web server queue was set to 75, rather than closer to the value of Max Application Concurrency. You can make similar adjustments when different components have different execution times.

  For example, in an application that spends 90% of its time in a complex servlet and only 10% of its time making a short Java database connectivity (JDBC) query, on average 10% of the servlets are using database connections at any time, so the database connection queue can be significantly smaller than the Web container queue. Conversely, if the majority of servlet execution time is spent making a complex query to a database, consider increasing the queue values at both the Web container and the data source. Always monitor the CPU and memory utilization for both the WebSphere Application Server and the database servers to verify that the CPU or memory are not overloaded.

Enterprise bean method invocation queuing

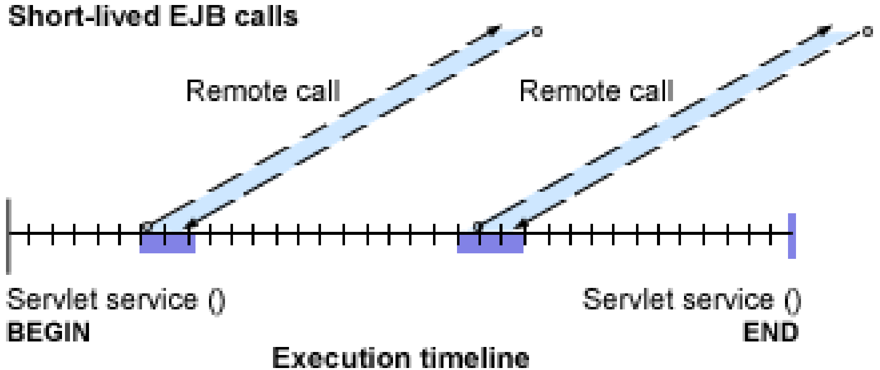
Method invocations to enterprise beans are queued only for remote clients that make the method call. An example of a remote client is an enterprise bean client running in a separate Java virtual machine (JVM) (another address space) from the enterprise bean. In contrast, no queuing occurs if the enterprise bean client, either a servlet or another enterprise bean, is installed in the same JVM on which the enterprise bean method runs and on the same thread of execution as the enterprise bean client.

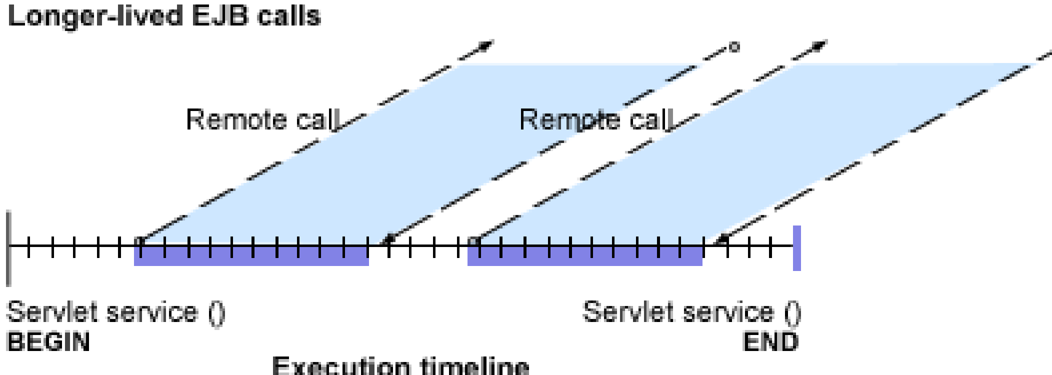
Remote enterprise beans communicate with the Remote Method Invocation over an Internet Inter-Orb Protocol (RMI-IIOP). Method invocations initiated over RMI-IIOP are processed by a server-side object request broker (ORB). The thread pool acts as a queue for incoming requests. However, if a remote method request is issued and there are no more available threads in the thread pool, a new thread is created. After the method request is completed, the thread is destroyed. Therefore, when the ORB processes remote method requests, the EJB container is an open queue, due to the use of unbounded threads. This graphic illustrates the two queuing options for enterprise beans.



Review this information about using enterprise bean queuing to improve performance:

- **Analyze the calling patterns of the enterprise bean client.**
  When you configure the thread pool, it is important to understand the calling patterns of the enterprise bean client. If a servlet is making a small number of calls to remote enterprise beans and each method call is relatively brief, consider setting the number of threads in the ORB thread pool to a value lower than the Web container thread pool size value.





  The degree to which the ORB thread pool value needs to be increased is a function of the number of clients (or sevlets) that simultaneously call enterprise beans, and of the duration of each method call. If the method calls are longer or the applications spend a lot of time in the ORB, consider making the ORB thread pool size equal to the Web container size. If the servlet makes only short-lived or quick calls to the ORB, servlets can potentially reuse the same ORB thread. In this case, the ORB thread pool can be as small as one-half of the thread pool size setting for the Web container.
- **Monitor the percentage of configured threads in use.**
  Tivoli Performance Viewer shows a metric called percent maxed, which is used to determine how often the configured threads are used. A value that is consistently in the double-digits indicates a possible bottleneck at the ORB. To remove the bottleneck, increase the number of threads.

Web server plug-in tuning tips

During normal operation, the backlog of connections pending to an application server can increase. Therefore, balancing the application server workload across a cluster can improve request response time.

In a distributed environment, you can use the MaxConnections server attribute in the Web server plug-in configuration file (plugin-cfg.xml) to define the maximum number of connections that can be pending to any of the application servers in the cluster. When this maximum number of connections is reached, the plug-in, when establishing connections, automatically skips that Application Server, and tries the next available application server. If no application servers are available, an HTTP 503 response code will be returned.

The capacity of the application servers in the network determines the value you specify for the MaxConnections attribute in the plugin-cfg.xml file. The ideal scenario is for all of the application servers in the network to be optimally utilized. For example, if you have the following environment:

- There are 10 WebSphere Application Server nodes in a cluster.
- All of these nodes host the same applications (that is, Application_1 and Application_2).
- This cluster of nodes is fronted by five IBM HTTP Servers.
- The IBM HTTP Servers get requests through a load balancer.
- Application_1 takes approximately 60 seconds to respond to a request.
- Application_2 takes approximately 1 second to respond to a request.

Depending on the request arrival pattern, all requests to Application_1 might be forwarded to two of the nodes, such as node_1 and node_2. If the arrival rate is faster than the processing rate, the number of pending requests to node_1 and node_2 can grow.

Eventually, node_1 and node_2 are busy and are not able to respond to future requests. It usually takes a long time to recover from this overloaded situation.

If you want to maintain 2500 connections, and optimally utilize the Application Servers in this example, set the MaxConnections attribute in the plugin-cfg.xml file to 50. (This value is arrived at by dividing the number of connections by the result of multiplying the number of Application Servers by the number of Web servers; in this example, 2500/(10x5)=50.)

The MaxConnections attribute works best with Web servers that follow the threading model instead of the process model, and only one process is started.

The IBM HTTP Server V1.3.x follows the process model. With the process model, a new process gets created to handle each connection from the Application Server, and typically, one process handles only one connection to the Application Server. Therefore, the MaxConnections attribute does not have much of an impact in restricting the number of concurrent requests to the Application Server.

The IBM HTTP Server V2.0.x follows the threading model. To prevent the IBM HTTP Server from starting more than one process, change the following properties in the Web server configuration file (httpd.conf) to the indicated values:

```
ServerLimit          1
ThreadLimit          4000
StartServers         1
MaxClients           1024
MinSpareThreads      1
MaxSpareThreads      1024
ThreadsPerChild      1024
MaxRequestsPerChild  0
```

Web services tuning tips

Web services performance is affected primarily by these characteristics of the XML documents that are sent to Web services:

- **Size** refers to the length of data elements in the XML document.
- **Complexity** refers to number of elements that the XML document contains.
- **Level of nesting** refers to objects or collections of objects that are defined within other objects in the XML document, as in this example:

```
<primaryObject>
  <groupObject>
    <singleObject/>
    <singleObject/>
  </groupObject>
</primaryObject>
```

In addition, a Web services engine contains three major pressure points that define the performance of Web services:

- **Parsing** (Input): When a request is received, the Web services engine parses the input. There are two major performance components in parsing:
  - scanner
  - symbol or name identification
- **XML-to-object deserialization** (Input): As the document is parsed the XML input is deserialized and converted into business objects that are presented as business object parameters to the Web services. The Web services provider, either a JavaBean provider or an enterprise bean provider, is not aware of its participation in a Web service.
- **Object-to-XML serialization** (Output): After the request is processed, reply is serialized into an XML document. Large documents or complex objects can affect output serialization.

Web Services Best Practices

- **Use WebSphere Application Server Web services instead of SOAP**
  The WebSphere Application Server Web services implementation performs better than the SOAP implementation based on Apache SOAP. WebSphere Application Server includes support for the Apache SOAP implementation so that you can run existing Web services applications.
- **Avoid large or complex XML documents**
  The performance of Web services is directly related to the size and complexity of the XML document that is transferred. As input documents increase in size or number of elements, they require more processing for both parsing and deserialization. As output documents increase in size or number of elements, they require more processing for serialization.
- **Avoid small, frequent requests**
  By definition, every Web services request is a remote request. These requests usually involve the Web container or Java Messaging Service (JMS) in addition to the XML overhead of parsing and deserialization. If you need to send or retrieve a 50K object that has 10 properties that are each 5K long, you can retrieve the object in several ways, such as:
  - As one 50K request
  - As 10 5K requests
  - As 50 1K requests

  Because of the overhead associated with the Web container or JMS, it is more efficient to transfer a single 50K request than several smaller requests.
- **Limit the level of nesting in XML documents**
  Increasing the level of object nesting results in an increase in the number of objects that are deserialized and created when a request is processed. An object that is composed only of primitive types or strings is processed more efficiently than a similar size object composed of deeply nested Java objects.
- **Use WebSphere Application Server custom serializers**
  The WebSphere Application Server Web services engine provides serialization and deserialization helpers that improve runtime performance for business objects. These custom serializer and deserializer helpers specifically describe an object's properties. As a result, the Web services runtime consumes fewer resources to obtain information about the object.
- **When possible, use literal encoding instead of SOAP encoding**
  Use literal encoding instead of SOAP encoding. Literal and SOAP encoding are alternate forms for encoding Web services requests and responses. SOAP encoding is the older and now less commonly used form of Web services encoding. Each element in the SOAP body includes the XML Schema definition type of the data element. SOAP encoding was needed to make the message self-defining. SOAP encoding with the embedded data types increases the amount of data transferred in the Web services request. We previously established the performance impact of XML message length. Figure 6 shows a comparison of data lengths for SOAP encoding versus document/literal for a typical workload.

- **Use descriptive but short property names**
  Web services is an XML text-based exchange protocol, and the names of variables and properties are included in the XML for the SOAP body portion of the message. Longer property names increase the size of the XML document.
- **When possible, use JavaBeans instead of session beans**
  J2EE Web services includes two server or provider types: JavaBean and session bean. Session beans provide well-defined interfaces that characterize a distributed service endpoint. However, there is extra overhead associated with session beans.

  Consider using a JavaBean instead of a session bean as the Web services provider. JavaBeans are not suitable in all cases; sometimes enterprise bean services are required for security and transactions. These factors are more important than the performance benefit of a JavaBean provider. However, if neither transactions nor security are needed for your Web service, a JavaBean provider might provide improved performance for your application.
- **Use 'Pass by Reference'**
  If you choose to use a session bean provider for your Web service, it is recommended that you use the WebSphere Application Server enterprise bean/IIOP **Pass by Reference** optimization, **No Local Copies**.

Hardware capacity and configuration

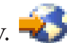
These parameters include considerations for selecting and configuring the hardware on which the application servers can run:

- **Disk speed**
  Disk speed and the number of disk arms can have a significant effect on application server performance in these cases:
  - Your application is heavily dependent on database support
  - Your application uses messaging extensively

  In these situations, it is recommended that you use disk I/O subsystems that are optimized for performance, such as a RAID array. Distribute the disk processing across as many disks as possible to avoid contention issues that typically occur with 1 or 2 disk systems. For more information about disk arms and how they can affect performance, see iSeries Disk Arm Requirements.
- **Processor speed**
  In the absence of other bottlenecks, increasing the processing power can improve throughput, response times, or both. On iSeries, processing power can be related to the Commercial Processing Workload (CPW) value of the system. For more information about CPW values, see the Performance Capabilities Reference Manual in the Performance Management Resource Library.
- **System memory**
  If a large number of page faults occur, performing these tasks might improve performance:
  - Increase the memory available to WebSphere Application Server.
  - Move WebSphere Application Server to another memory pool.
  - Remove jobs from the WebSphere Application Server memory pool.

  To determine the current page fault level, run the Work with System Status (WRKSYSSTS) command from an OS/400 command line. For information about the minimum memory requirements, see iSeries and AS/400 hardware requirements.
- **Storage pool activity levels**
  Verify that the activity levels for storage pools are sufficient. Increasing these values can prevent threads from transitioning into the ineligible condition.
  - To modify the activity level for the storage pool in which you are running WebSphere Application Server, run the Work with System Status (WRKSYSSTS) command from an OS/400 command line:
    ```
    WRKSYSSTS ASTLVL(*INTERMED)
    ```
  - Set the system value QMAXACTLVL to a value equal to or greater than the total activity level for all pools, or *NOMAX.

1. Run the Work with System Value (WRKSYSVAL) command from an OS/400 comand line:
   `WRKSYSVAL SYSVAL(QMAXACTLVL)`
2. Adjust the value in the **Max Active** column.

- **Networks**
  Run network cards and network switches at full duplex. Running at half duplex decreases performance. Verify that the network speed can accommodate the required throughput. On 10/100 Ethernet networks, verify that 100MB is in use. You might also want to monitor the IOP utilization. For information about IOP utilization, see the Performance Capabilities Reference in the Performance Management Resource Library. 

Java virtual machine tuning parameters

Because the application server is a Java process, it requires a Java virtual machine (JVM) to run, and to support the Java applications running on it. For more information about JVM settings, see Java virtual machine settings.

For additional tuning information, see “Java memory tuning tips” on page 65. If your application experiences slow response times at startup or first touch, you may want to consider using the Java user classloader cache. For more information, see Java cache for user classloaders in the *Programming* topic.

You can adjust these settings that determine how the system uses the JVM:

- **Class garbage collection (-Xnoclassgc)**
  - Description: This argument disables class garbage collection so that your applications can reuse classes more easily. You can monitor garbage collection using the -verbosegc configuration setting because its output includes class garbage collection statistics.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Process Definition**.
    5. On the **Process Definition** page, click **Java Virtual Machine**.
    6. In the **Generic JVM arguments** field, type `-Xnoclassgc`.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  - Default value: By default, class garbage collection is enabled.
  - Recommended value: Do not disable class garbage collection.
- **Initial heap size**
  - Description: The initial heap size specifies how often garbage collection runs, and can have a significant effect on performance. For more information, see Tuning Garbage Collection for Java(TM) and WebSphere on iSeries.
  - How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Process Definition**.
    5. On the **Process Definition** page, click **Java Virtual Machine**.
    6. In the **Initial Heap Size** field, specify a value.

7. Click **Apply** or **OK**.
8. Save the configuration.
9. Stop and restart the application server.
– Default value: For iSeries, the default value is 96.
– Recommended value: 96MB per processor

- **Maximum heap size**
  – Description: This parameter specifies the maximum heap size available to the JVM code, in megabytes.
  – How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Process Definition**.
    5. On the **Process Definition** page, click **Java Virtual Machine**.
    6. The value is displayed in the **Maximum Heap Size** field.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  – Default value: For iSeries, the default value is 0. A value of 0 specifies that there is no maximum value.
  – Recommended value: It is recommended that you do not change the maximum heap size. When the maximum heap size triggers a garbage collection cycle, the iSeries JVM's garbage collection stops operating asynchronously. As a result, the application server cannot process user threads until the garbage collection cycle ends, and performance is significantly lower.
- **Just-In-Time (JIT) compiler**
  – Description: A Just-In-Time (JIT) compiler is a platform-specific compiler that generates machine instructions for each method as needed. For more information, see Using the Just-In-Time compiler and Just-In-Time compiler in the *IBM Developer Kit for Java* topic.
  – How to view or set:
    1. Start the administrative console.
    2. In the topology tree, expand **Servers** and click **Application Servers**.
    3. Click the name of the application server that you want to configure.
    4. Click **Process Definition**.
    5. On the **Process Definition** page, click **Java Virtual Machine**.
    6. If you want to disable JIT, select the checkbox for **Disable JIT**.
    7. Click **Apply** or **OK**.
    8. Save the configuration.
    9. Stop and restart the application server.
  – Default value: By default, JIT is enabled.
  – Recommended value: It is recommended that you do not disable the JIT compiler. The os400.jit.mmi.threshold can have a significant effect on performance. For more information about the JIT compiler and the os400.jit.mmi.threshold property, see Just-In-Time compiler in the *IBM Developer Kit for Java* topic.

Java memory tuning tips

Enterprise applications written in the Java language involve complex object relationships and utilize large numbers of objects. Although the Java language automatically manages memory associated with object life cycles, understanding the application usage patterns for objects is important. For more information, see WebSphere and Java tuning tips.

- Verify that the application is not creating a large number of short-lived objects.
- Verify that the application is not leaking objects.
- Verify that the Java heap parameters are set properly to handle a given object usage pattern.

Understanding the effect of garbage collection is necessary to apply these management techniques.

The Java garbage collection bottleneck

Examining Java garbage collection can help you understand how the application is utilizing memory. Because Java provides garbage collection, your application does not need to manage server memory. As a result, Java applications are more robust than applications written in languages that do not provide garbage collection. This robustness applies as long as the application is not over-utilizing objects.

Garbage collection normally consumes from 5% to 20% of total execution time of a properly functioning application. If you do not manage garbage collection, it can have a significant negative impact on application performance, especially when running on symmetric multiprocessing (SMP) server machines.

The OS/400 JVM uses concurrent (asynchronous) garbage collection. This type of garbage collection results in shorter pause times and allows application threads to continue processing requests during the garbage collection cycle.

Garbage collection in the OS/400 JVM is controlled by the heap size settings. link in min and max heap parms on tuning page). The initial heap size is a threshold that triggers new garbage collection cycles. If the initial heap size is 10 MB, for example, then a new collection cycle is triggered as soon as the JVM detects that 10 MB have been allocated since the last collection cycle. Smaller heap sizes result in more frequent garbage collections than larger heap sizes. If the maximum heap size is reached, the garbage collector stops operating asynchronously, and user threads are forced to wait for collection cycles to complete. This situation has a significant negative impact on performance. A maximum heap size of 0 (*NOMAX) assures that garbage collection operates asynchonously at all times. For more information about tuning garbage collection with the JVM heap settings, see “Java virtual machine tuning parameters” on page 63.

The garbage collection gauge

You can use garbage collection to evaluate application performance health. Monitoring garbage collection when the server is under a fixed workload can help you determine if the application is creating several short-lived objects and can detect the presence of memory leaks.

You can monitor garbage collection statistics with any of these tools:

- Object statistics in Tivoli Performance Viewer
  For information about these statistics, see the last three entries in the table in Java virtual machine data counters.
- The -verbosegc JVM configuration setting
  If you specify this setting, garbage collection generates verbose output.

  Note: The -verbosegc format is not standardized between different JVMs or release levels.
- The Dump Java Virtual Machine (DMPJVM) command
  This command dumps JVM information for a specified job.

- Heap Analysis Tools for Java(TM)
  This tool is a component of the iDoctor for iSeries suite of performance monitoring tools. The Heap Analysis Tools component performs Java application heap analysis and object create profiling (size and identification) over time. This tool is sometimes called Java Watcher or Heap Analyzer. For more information, about iDoctor for iSeries, see iDoctor for iSeries.
- Performance Explorer (PEX)
  You can use a Performance Explorer (PEX) trace to determine how much CPU is being used by the garbage collector. For detailed instructions, see Tuning Garbage Collection for Java(TM) and WebSphere on iSeries.

To obtain meaningful statistics, run the application under a fixed workload until the application state is steady. It usually takes several minutes to reach a steady state.

Detecting large numbers of short-lived objects

To determine if an application is creating a large number of short-lived objects, monitor the JVM run time counters in Tivoli Performance Viewer. For information about enabling the Java virtual machine profiler interface (JVMPI) counters, see "Enable Java Virtual Machine Profiler Interface (JVMPI) data reporting" on page 19.

You can also use these tools to monitor JVM object creation:
- The DMPJVM (Dump Java Virtual Machine) command
  The DMPJVM command dumps information about the JVM for a specified job.
- The ANZJVM (Analyze Java Virtual Machine) command
  The ANZJVM command collects information about the Java Virtual Machine (JVM) for a specified job. This command is available in OS/400 V5R2 and later.
- The Performance Trace Data Visualizer (PTDV)

The best result for the average time between garbage collections is at least 5 to 6 times the average duration of a single garbage collection cycle. If the average time is shorter, the application is spending more than 15% of its time in garbage collection.

If the information indicates a garbage collection bottleneck, there are two ways to clear the bottleneck. The most efficient way to optimize the application is to implement object caches and pools. Use a Java profiler to determine which objects to cache. If you can not optimize the application, you can add memory, processors, and clones. Additional memory allows each clone to maintain a reasonable heap size. Additional processors allow the clones to run in parallel.

Detecting memory leaks

Memory leaks in the Java language are a significant contributor to garbage collection bottlenecks. Memory leaks are more damaging than memory overuse, because a memory leak ultimately leads to system instability. Over time, there is typically an increase in paging and garbage collection times. Garbage collection times increase until the heap is too large to fit into memory, paging rates increase, and eventually garbage collections are forced into synchronous mode. As a result, threads that are waiting for memory allocation are stopped. From a client's point of view, the application stops processing requests. Clients might also receive java.lang.OutOfMemoryError exceptions.

Memory leaks occur when an unused object has references that are never freed. Memory leaks most commonly occur in collection classes, such as Hashtable because the table always has a reference to the object, even after real references are deleted.

High workload often causes applications to perform poorly after deployment in the production environment. This is especially true for leaking applications where the high workload accelerates the magnification of the leakage and the heap size grows too large for the garbage collector to manage.

The goal of memory leak testing is to magnify numbers. Memory leaks are measured in terms of the amount of bytes or kilobytes that garbage collection cannot collect. The delicate task is to differentiate these amounts between expected sizes of useful and unusable memory. This task is achieved more easily if the numbers are magnified, resulting in larger gaps and easier identification of inconsistencies. The following list contains important conclusions about memory leaks:

- **Long-running test**
  Memory leak problems can manifest only after a period of time. Therefore, memory leaks are found easily during long-running tests. Short running tests can lead to false alarms. It is sometimes difficult to know when a memory leak is occurring in the Java language, especially when memory usage has seemingly increased either abruptly or monotonically in a given period of time. The reason it is hard to detect a memory leak is that these kinds of increases can be valid or might be the intention of the developer. You can learn how to differentiate the delayed use of objects from completely unused objects by running applications for a longer period of time. Long-running application testing gives you higher confidence for whether the delayed use of objects is actually occurring.
- **Repetitive test**
  In many cases, memory leak problems occur by successive repetitions of the same test case. The goal of memory leak testing is to establish a big gap between unusable memory and used memory in terms of their relative sizes. By repeating the same scenario over and over again, the gap is multiplied in a very progressive way. This testing helps if the number of leaks caused by the execution of a test case is so minimal that it is hardly noticeable in one run.

  You can use repetitive tests at the system level or module level. The advantage with modular testing is better control. When a module is designed to keep the private module without creating external side effects such as memory usage, testing for memory leaks is easier. First, the memory usage before running the module is recorded. Then, a fixed set of test cases are run repeatedly. At the end of the test run, the current memory usage is recorded and checked for significant changes.
- **Concurrency test**
  Some memory leak problems can occur only when there are several threads running in the application. Unfortunately, synchronization points are very susceptible to memory leaks because of the added complication in the program logic. Careless programming can lead to kept or unreleased references. The incident of memory leaks is often facilitated or accelerated by increased concurrency in the system. The most common way to increase concurrency is to increase the number of clients in the test driver.

  Consider the following points when choosing which test cases to use for memory leak testing:
  - A good test case exercises areas of the application where objects are created. Most of the time, knowledge of the application is required. A description of the scenario can suggest creation of data spaces, such as adding a new record, creating an HTTP session, performing a transaction and searching a record.
  - Look at areas where collections of objects are used. Typically, memory leaks are composed of objects within the same class. Also, collection classes such as Vector and Hashtable are common places where references to objects are implicitly stored by calling corresponding insertion methods. For example, the get method of a Hashtable object does not remove its reference to the retrieved object.

You can use these tools to detect memory leaks:

- “Monitor performance with Tivoli Performance Viewer” on page 13
  For more information and examples of using Tivoli Performance Viewer to detect memory leaks, see Tuning Garbage Collection for Java(TM) and WebSphere on iSeries.
- The DMPJVM (Dump Java Virtual Machine) command
  The DMPJVM command dumps information about the JVM for a specified job.

- The ANZJVM (Analyze Java Virtual Machine) command
  The ANZJVM command collects information about the Java Virtual Machine (JVM) for a specified job. This command is available in OS/400 V5R2 and later.
- Heap Analysis Tools for Java(TM)
  This tool is a component of the iDoctor for iSeries suite of performance monitoring tools. The Heap Analysis Tools component performs Java application heap analysis and object create profiling (size and identification) over time. This tool is sometimes called Java Watcher or Heap Analyzer.

For best results, follow these guidelines:

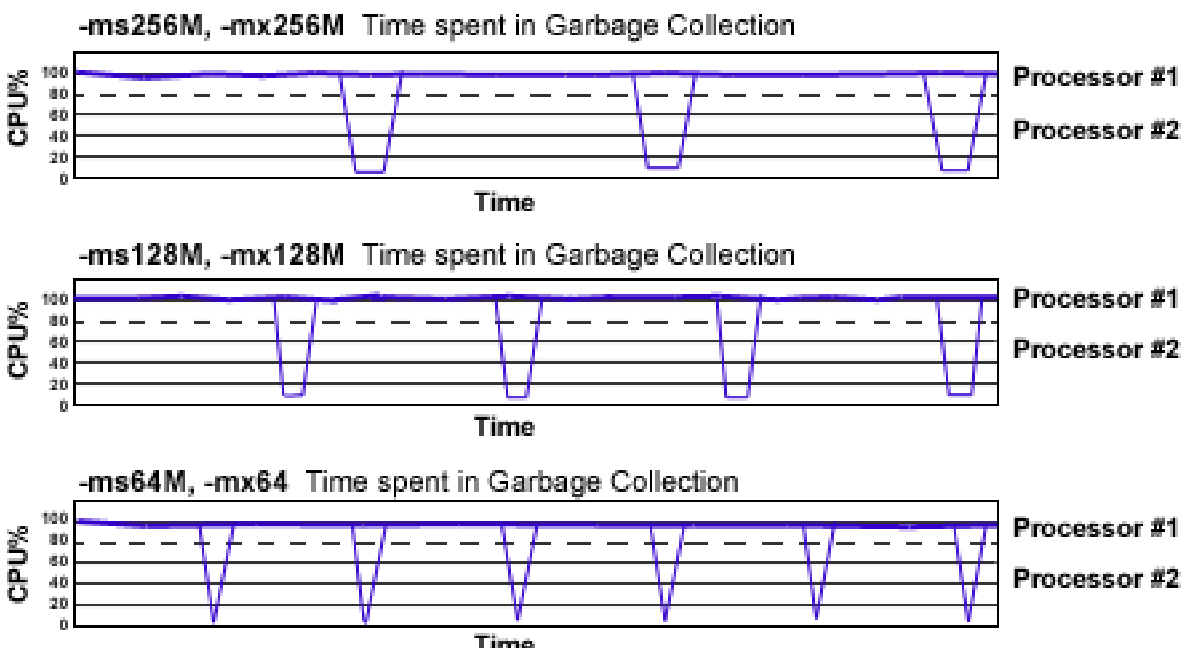
- Use the tools to take a series of readings of the number of objects in the heap. Allowing at least 10 to 20 garbage collection cycles between each reading. You can compare the results of each reading to see if there are classes with a monotonically increasing count of objects.
- Repeat experiments with increasing duration, like 1000, 2000, and 4000-page requests. If the application uses several objects that are larger than 64KB, the total heap size may decrease after a garbage collection cycle. The JVM reuses the heap space for smaller objects, and only releases that space after long periods of idle activity. If the heap size continually increases and never reaches a steady state, there might be a memory leak.
- Look at the difference between the number of objects allocated and the number of objects freed. If the gap between the two increases over time, there is a memory leak. In most cases, determining object counts by class is the most useful way to detect leaks with the OS/400 JVM.

Java heap parameters

- **Initial heap size**

  The Java heap parameters also influence the behavior of garbage collection. Because a large heap takes longer to fill, the application runs longer before a garbage collection occurs. For more information about heap settings, see "Java virtual machine tuning parameters" on page 63.

  When tuning a production system where the working set size of the Java application is not understood, it is recommended that you set the initial heap size to 96MB per processor. The total heap size in an OS/400 JVM can be approximated as the sum of the amount of live (in use) heap space at the end of the last garbage collection plus the initial heap size.



The illustration represents three CPU profiles, each running a fixed workload with a varying initial Java heap size. In the middle profile, the initial size is set to 128MB. Four garbage collections occur. The total time in garbage collection is about 15% of the total run. When the initial heap size is doubled to 256MB, as in the top profile, the length of the work time increases between garbage collections. Only three garbage collections occur, but the length of each garbage collection is also increased. In the third profile, the heap size is reduced to 64MB and exhibits the opposite effect. With a smaller heap size, both the time between garbage collections and the time for each garbage collection are shorter.

This example shows that the total time in garbage collection is approximately the same in all cases. However, in most cases, setting a smaller initial heap size results in more total time spent in garbage collection, especially if the initial heap size is small compared to the pace of object allocation. If the initial heap size is too small the garbage collector runs almost continuously.

Run a series of test experiments that vary the initial Java heap settings. For example, run experiments with 128MB, 192MB, 256MB, and 320MB. During each experiment, monitor the total memory usage. When all of the runs are finished, compare these statistics:

– Number of garbage collection calls

– Average duration of a single garbage collection call

– Ratio between the length of a single garbage collection call and the average time between calls

If the application is not over-utilizing objects and has no memory leaks, the state of steady memory utilization is reached. Garbage collection also occurs less frequently and for short duration.

Note that unlike other JVM implementations, a large amount of heap free space is not generally a concern for the OS/400 JVM.

- **Maximum heap size**

The maximum heap size can affect application performance. This value specifies the maximum amount of object space the garbage collected heap can consume. If the maximum heap size is too small, performance might degrade significantly, or the application might receive out of memory errors when the maximum heap size is reached. Due to the complexity of determining a correct value for the maximum heap size, a value of 0 (meaning there is no size limit) is recommended unless an absolute limit on the object space for the garbage collected heap size is required.

In a situation where an absolute limit for the garbage collected heap is required, the value specified should be large enough so that performance is not negatively affected. To determine an appropriate value, run your application under a heavy load with a maximum heap value of 0. Determine the maximum size of the garbage collected heap for the JVM using DMPJVM or iDoctor. The smallest acceptable value for the maximum heap size is 125 percent of the garbage collected heap size. This value is a reasonable estimate for your garbage collected heap working set size. You can specify a larger value for the maximum heap size without affecting performance, and it is recommended that you set the largest possible value based on the resource restrictions of the JVM or the limitations of the system configuration.

After you determine an appropriate value for the maximum heap size, you might need to set up or adjust the pool in which the JVM runs. By default, WebSphere Application Server jobs run in the base system pool (storage pool 2 as shown by WRKSYSSTS), but you can specify a different pool. The maximum heap size should not be set larger than 125 percent of the size of the pool in which the JVM is running. It is recommended that you run the JVM in its own memory pool with the memory permanently assigned to that pool, if possible.

If the performance adjuster is set to adjust the memory pools (that is, the system value QPFRADJ is set to a value other than 0), it is recommended that you specify a minimum size for the pool using WRKSHRPOOL. The minimum size should be approximately equal to your garbage collected heap working set size. Setting a correct maximum heap size and properly configuring the memory pool can prevent a JVM with a memory leak from consuming system resources, but still offers excellent performance.

When a JVM must run in a shared pool, it is more difficult to determine an appropriate value for the maximum heap size. Other jobs running in the pool can cause the garbage collected heap pages to be aged out of the pool. If the garbage collected heap pages are aged out of the pool, the garbage collector must fault the pages back into the pool on the next garbage collection cycle because it needs to access

all of the pages in the garbage collected heap. Because the OS/400 JVM does not stop all of the JVM threads to clean the heap, excessive page faulting causes the garbage collector to slow down and the garbage collected heap to grow. Instead the size of the heap is increased and threads continue to run.

This heap growth is an artificial inflation of the garbage collected heap working set size, and must be considered if you want to specify a maximum heap value. When a small amount of artifical inflation occurs, the garbage collector reduces the size of the heap over time if the space remains unused and the activity in the pool returns to a steady state. However, in a shared pool, you might experience problems if the maximum heap size is set incorrectly

– If the maximum heap size is too small, artifical inflation can result in severe performance degradation or system failure if the JVM throws an out of memory error.
– If the maximum heap size is set too large the garbage collector might reach a point where it is unable to recover the artificial inflation of the garbage collected heap. In this case, performance is also negatively affected. A value that is too large might not be able to prevent a JVM failure, but it can prevent a run-away JVM from consuming excessive amounts of system resources.

If you want to determine the proper value for the maximum heap size, you must run multiple tests, because the appropriate value is different for each configuration or workload combination. If you want to prevent a run-away JVM, set the maximum heap size larger than you expect the heap to grow, but not so large that it affects the performance of the rest of the machine.

If you must set the maximum heap size to guarantee that the heap size does not exceed a given level, specify an initial heap size that is 80-90% smaller than the maximum heap size.

Web server tuning parameters

WebSphere Application Server provides plug-ins for several Web server brands and versions. If you are running your Web server on a non-iSeries platform, see the product documentation for performance tuning information. For additional information, refer to Chapter 6 of the Performance Capabilities Reference Manual. This manual is available in the Performance Management Resource Library.

The IBM HTTP Server (powered by Apache) is a multi-process, multi-threaded server. For IBM HTTP Server for iSeries, you can adjust these settings:

- **Access logs**
  – Description: The access logs record all incoming HTTP requests. Logging can degrade performance. On iSeries, overhead is minimized, because logging occurs in a separate process from the Web server function.
  – How to view or set:
    1. Open the IBM HTTP Server httpd.conf file, located in the /QIBM/ProdData/HTTPA/conf directory.
    2. Search for lines with the text CustomLog.
    3. To enable a custom access log, remove the hash mark (#) at the beginning of the line.
    4. Save and close the httpd.conf file.
    5. Stop and restart the IBM HTTP Server.
  – Default value: By default, the access log is disabled.
  – Recommended value: Do not enable the access logs.
- **ThreadsPerChild**
  – Description: This directive specifies the maximum number of concurrent client requests that the server processes at any time. The Web server uses one thread for each request that it processes. This value does not represent the number of active clients.
  – How to view or set: Edit or view the ThreadsPerChild directive in the IBM HTTP Server httpd.conf file.
  – Default value: 40

- Recommended value: It is recommended that you use the default value, and only increase this value if necessary.

- **ListenBackLog**
  - Description: This parameter sets the length of a pending connections queue. When several clients request connections to the IBM HTTP Server, and all threads are in use, a queue exists to hold additional client requests. However, if you use the default Fast Response Cache Accelerator (FRCA) feature, the ListenBackLog directive is not used, because FRCA uses its own internal queue.
  - How to view or set: For non-FRCA: Edit or view the ListenBackLog directive in the IBM HTTP Server httpd.conf file.
  - Default value: For IBM HTTP Server 1.3.26: 1024 with FRCA enabled, 511 with FRCA disabled
  - Recommended value: Use the default value.

Database tuning parameters

For tuning information for DB2 UDB for iSeries, see these resources:

- Monitor and tune database performance in the *Database* topic.
- Chapter 4 of the Performance Capabilities Reference. Links to several editions of the Performance Capabilities Reference are listed in the Performance Management Resource Library.

You might also want to adjust the QSQSRVR prestart job settings for your system. On iSeries, QSQSRVR jobs process Java database access (JDBC). By default, five QSQSRVR jobs are initially active. When fewer than two QSQSRVR jobs are unused, two more jobs are created. An application that establishes a large number of database connections over a short period of time may be able to create connections more quickly if these values are increased. To increase the initial number of jobs, threshold, and additional number of jobs values, run this command on an OS/400 command line:

```
CHGPJE SBSD(QSYS/QSYSWRK) PGM(QSYS/QSQSRVR)
```

Do not start more QSQSRVR jobs than your application requires, because there is some overhead associated with having QSQSRVR jobs active, even if they are not being used.

If you use a different database, refer to that product's documentation.

TCP/IP buffer sizes

WebSphere Application Server uses the TCP/IP sockets communication mechanism extensively. For a TCP/IP socket connection, the send and receive buffer sizes define the receive window. The receive window specifies the amount of data that can be sent and not received before the send is interrupted. If too much data is sent, it overruns the buffer and interrupts the transfer. The mechanism that controls data transfer interruptions is referred to as flow control. If the receive window size for TCP/IP buffers is too small, the receive window buffer is frequently overrun, and the flow control mechanism stops the data transfer until the receive buffer is empty.

Flow control can consume a significant amount of CPU time and result in additional network latency as a result of data transfer interruptions. It is recommended that you increase buffer sizes avoid flow control under normal operating conditions. A larger buffer size reduces the potential for flow control to occur, and results in improved CPU utilization. However, a large buffer size can have a negative effect on performance in some cases. If the TCP/IP buffers are too large and applications are not processing data fast enough, paging can increase. The goal is to specify a value large enough to avoid flow control, but not so large that the buffer accumulates more data than the system can process.

The default buffer size is 8KB. The maximum size is 8MB. The optimal buffer size depends on several network environment factors including types of switches and systems, acknowledgment timing, error

rates and network topology, memory size, and data transfer size. When data transfer size is extremely large, you might want to set the buffer sizes up to the maximum value to improve throughput, reduce the occurrence of flow control, and reduce CPU cost.

Buffer sizes for the socket connections between the Web server and WebSphere Application Server are set at 64KB. In most cases this value is adequate.

Flow control can be an issue when an application uses either the IBM Developer Kit for Java(TM) JDBC driver or the IBM Toolbox for Java JDBC driver to access a remote database. If the data transfers are large, flow control can consume a large amount of CPU time. If you use the IBM Toolbox for Java JDBC driver, you can use custom properties to configure the buffer sizes for each data source. It is recommended that you specify large buffer sizes, such as 1MB.

Some system-wide settings can override the default 8KB buffer size for sockets. With some applications, such as WebSphere Commerce Suite, a buffer size of 180KB reduces flow control and typically does not adversely affect paging. The optimal value is dependent on specific system characteristics. You might need to try several values before you determine the ideal buffer size for your system. To change the system wide value follow these steps:
1. Run the Change TCP/IP Attributes (CHGTCPA) command on an OS/400 command line.
2. On the **Change TCP/IP Attributes** display, press **F4**. The buffer sizes are displayed as the TCP receive and send buffer size.
3. Specify new values.
4. Save your changes.
5. Recycle TCP/IP.
6. Monitor CPU and paging rates to determine if they are within recommended system guidelines.

Repeat this process until you determine the ideal buffer size.

For more information about TCP/IP performance, see Chapter 5 of the Performance Capabilities Reference. Links to several editions of the Performance Capabilities Reference are listed in the Performance Management Resource Library.

Application assembly performance checklist

Application assembly tools are used to build J2EE components and modules into J2EE applications. Application assembly consists of defining application components and their attributes including enterprise beans, servlets and resource references. Many of these application configuration settings and attributes play an important role in the run-time performance of the deployed application. Use this information as a check list of important parameters and advice for finding optimal settings:
- EJB module settings
  - Entity bean settings
    - Bean Cache - Activate at
    - Bean Cache - Load at
  - Method extensions settings
    - Isolation level
    - Access intent
  - Container transactions assembly settings
- Web modules assembly settings
  - Distributable

– Reload interval
– Reload enabled

- Web component settings
  – Load on startup

For information about JSP compilation, see these topics in *Application development*:

- Pre-touch tool for compiling and loading JSP files - The JSP pre-touch tool can enhance application server performance by causing all JSP files to be compiled when your application server starts.
- JavaServer Pages (JSP) - This page includes general information about JSP files, and provides links to information about JSP batch compilation, disabling run-time compilation, and reducing JSP compile time.

Troubleshoot performance

This table describes some common performance problems and the action that you can take to resolve them.

Problem	User action
ORB: Response time and throughput indicate that enterprise bean requests with shorter processing times are being denied adequate access to threads in ORB thread pool.	Set the Logical Pool Distribution (LPD) mechanism for the ORB service. See Object Request Broker service custom properties for more information.
Under a load, client requests do not arrive at the Web server because they time out or are rejected.	For IBM HTTP Server for iSeries (powered by Apache), set the ListenBackLog parameter.
The Percent Maxed metric from Tivoli Performance Viewer indicates that the Web container thread pool is too large or too small. For information about the Percent Maxed metric, see **Thread pool** on PMI data counters (page 12).	Set the Thread pool Maximum size for the Web container.
Netstat shows too many TIME_WAIT state sockets for the application server's internal HTTP port (9080 is the default port).	Set the HTTP transports MaxKeepAliveConnections and HTTP transports MaxKeepAliveRequests properties. See Set custom properties for an HTTP transport for more information.
Disk arm utilization is too high, due to excessive paging.	Verify that there is enough memory in the pool in which WebSphere Application Server is running, and that the Java virtual machine heap is not leaking. For more information, see "Java virtual machine tuning parameters" on page 63 and "Java memory tuning tips" on page 65.
The Percent Used metric for a data source connection pool from Tivoli Performance Viewer indicates the pool size is too large. For information about the Percent Used metric, see **JDBC connection pools** on PMI data counters (page 11).	Set the Maximum connection pool and Minimum connection pool properties. See Connection pool settings for more information.
The Prepared Statement Discards metric from Tivoli Performance Viewer indicates that the data source statement cache size is too small. For information about the Prepared Statement Discards metric, see **JDBC connection pools** on PMI data counters (page 11).	Set the Statement cache size property. See Data source settings for more information.
The Percent Maxed metric from Tivoli Performance Viewer indicates that the Object Request Broker thread pool is too small. For information about the Percent Maxed metric, see **Thread pool** on PMI data counters (page 12).	Configure "Enterprise bean method invocation queuing" on page 58.

Problem	User action
The server is unresponsive and paging occurs.	Check for memory leaks and verify that enough memory is available. See “Java memory tuning tips” on page 65 for more information.
Throughput, response time and scalability are undesirable.	Use the dynamic cache service.

Performance advisors

IBM WebSphere Application Server includes two performance advisors to help tune systems for optimal performance. These performance advisors use the PMI data to suggest configuration changes based on which settings might need to be adjusted for your environment. For more information about PMI data, see “Performance Monitoring Infrastructure (PMI)” on page 8.

The Runtime Performance Advisor runs in the application server process, while the other advisor runs in the Tivoli Performance Viewer. The following table outlines the difference between the two advisors:

	Runtime Performance Advisor	Performance Advisor in Tivoli Performance Viewer
Location of execution	Application server	Tivoli Performance Viewer client
Location of tool	Administrative console	Tivoli Performance Viewer
Output	SystemOut.log file and WebSphere run-time messages in WebSphere status panel in the administrative console	Tivoli Performance Viewer graphical user interface
Frequency of operation	Every 10 seconds in background	When you select refresh in Tivoli Performance Viewer
Types of advice	• ORB service thread pools • Web container thread pools • Connection pool size • Persisted session size and time • Prepared statement cache size • Session cache size	• ORB service thread pools • Web container thread pools • Connection pool size • Persisted session size and time • Prepared statement cache size • Session cache size • Dynamic cache size • JVM heap size • DB2 Performance Configuration Wizard

For more detailed information about the performance advisors, see these topics:

“Use the Performance Advisor in Tivoli Performance Viewer” on page 75
The Performance Advisor in Tivoli Performance Viewer provides information to help you optimize performance. This topic describes how to configure and use the Performance Advisor.

“Use the Runtime Performance Advisor” on page 76
The Runtime Performance Advisor provides information to help you optimize performance. This topic describes how to configure and use the Runtime Performance Advisor.

Use the Performance Advisor in Tivoli Performance Viewer

The Performance Advisor in Tivoli Performance Viewer analyzes collected PMI data and provides advice to help tune systems for optimal performance. The Performance Advisor in Tivoli Performance Advisor provides more extensive advice than the Runtime Performance Advisor provides. For example, the Performance Advisor in Tivoli Performance Viewer provides advice on setting the dynamic cache size and setting the Java virtual machine heap size.

For more information about Tivoli Performance Viewer, see "Monitor performance with Tivoli Performance Viewer" on page 13. For more information about PMI, see "Performance Monitoring Infrastructure (PMI)" on page 8.

To enable and run the Performance Advisor, follow these steps:

1. "Enable performance monitoring services for application servers" on page 20 and restart the application server.
2. (Optional) If you are running WebSphere Application Server Network Deployment, you must also "Enable performance monitoring services for node agents" on page 20 and restart the node agent.
3. Enable data collection:
   - "Enable performance data collection with the administrative console" on page 22.
   - "Enable performance monitoring services with wsadmin" on page 21.

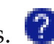
   The performance monitoring settings are automatically detected. You do not need to restart the application server.
4. "Start Tivoli Performance Viewer" on page 14.
5. Simulate a production level load. If you are using the Performance Advisor in a test environment, or doing any other performance tuning, simulate a realistic production load for your application. The application should run this load without errors. This simulation includes numbers of concurrent users typical of peak periods, and drives system resources such as CPU and memory to the levels expected in production. Some advice may not be accurate if sufficient load is not generated.
6. Select the the Advisor icon to display tuning advice. Double-click a message for details. Because PMI data is taken over an interval of time and averaged to provide advice, details within the advice message appear as averages.
7. (Optional) Refresh data. You can refresh data in two ways:
   - Select your application server under **Viewer —>** ***node_name***, then click **Refresh**. Tivoli Performance Viewer queries the server for new PMI and product configuration information.
   - Select your application server under **Performance Advisor —>** ***node_name***, then click **Refresh**. Tivoli Performance viewer refreshes advice that is provided at a single instant in time, but does not query the server for new PMI and product configuration information.
8. Update the product configuration based on the advice.
   Because Tivoli Performance Viewer refreshes advice at a single instant in time, take the advice from the peak load time. Although the performance advisor attempts to distinguish between loaded and idle conditions, it might provide misleading advice if it is enabled when the system is ramping up or down. This result is especially likely during short tests. Although the advice helps in most configurations, there might be situations where the advice hinders performance. Due to these conditions, advice is not guaranteed. Therefore, test the environment with the updated configuration to verify that it functions and performs well.

For information about the report, see Performance Advisor Report in Tivoli Performance Viewer.

Use the Runtime Performance Advisor

The Runtime Performance Advisor provides advice to help tune systems for optimal performance. The Runtime Performance Advisor uses Performance Monitoring Infrastructure (PMI) data to provide recommendations for performance tuning. For more information about PMI, see "Performance Monitoring Infrastructure (PMI)" on page 8.

This advisor runs in the JVM of the application server, and periodically checks for inefficient settings. Recommendations are provided in the form of standard product warning messages. These recommendations are displayed both as warnings in the administrative console under WebSphere Runtime Messages and as text in the application server SystemOut.log file. The Runtime Performance Advisor has minimal system performance impact.

For more information about configuration settings for the Runtime Performance Advisor, see Runtime Performance Advisor configuration settings. 

The Runtime Performance Advisor enables the appropriate monitoring counter levels for all enabled advice. If there are specific counters that you do not want to monitor, disable the corresponding advice in the Runtime Performance Advisor, and disable unwanted counters.

Note: Before you enable the Runtime Performance Advisor, you must "Enable performance monitoring services for application servers" on page 20 and restart the application server. For WebSphere Application Server Network Deployment, you must also "Enable performance monitoring services for node agents" on page 20 and restart the node agent.

To use the Runtime Performance Advisor, follow these steps:

1. Start the administrative console.
2. In the administrative console topology tree, expand **Servers** and click **Application Servers**.
3. Click the name of the server for which you want to configure the Runtime Performance Advisor.
4. Click **Runtime Performance Advisor Configuration**.
5. On the **Configuration** tab, configure these settings:
   - Select the **Calculation Interval**. PMI data is taken over an interval of time and averaged to provide advice. The interval specifies the length of the time over which data is taken for this advice. Therefore, details within the advice messages appear as averages over this interval.
   - Select the **Maximum Warning Sequence**. The maximum warning sequence refers to the number of consecutive warnings issued before the threshold is updated. For example, if the maximum warning sequence is set to 3, then the advisor only sends three warnings to indicate that the prepared statement cache is overflowing. After that, a new alert is sent only if the rate of discards exceeds the new threshold setting.
   - Specify **Number of Processors**. Select the appropriate settings for your system configuration to ensure accurate advice.

     Note: A limited number of values is available. If the number of processors in your WebSphere Application Server partition is not listed, select the next highest number.
6. Click **Apply**.
7. Save the configuration.
8. In the administrative console topology tree, expand **Servers** and click **Application Servers**.
9. Click the name of the server for which you want to configure the Runtime Performance Advisor.
10. Click **Runtime Performance Advisor Configuration**.
11. Click the **Runtime** tab.
12. On the **Runtime** tab, configure these settings:

- Select the **Calculation Interval**. PMI data is taken over an interval of time and averaged to provide advice. The interval specifies the length of the time over which data is taken for this advice. Therefore, details within the advice messages appear as averages over this interval.
- Select the **Maximum Warning Sequence**. The maximum warning sequence refers to the number of consecutive warnings issued before the threshold is updated. For example, if the maximum warning sequence is set to 3, then the advisor only sends three warnings to indicate that the prepared statement cache is overflowing. After that, a new alert is sent only if the rate of discards exceeds the new threshold setting.
- Specify **Number of Processors**. Select the appropriate settings for your system configuration to ensure accurate advice.

  Note: A limited number of values is available. If the number of processors in your WebSphere Application Server partition is not listed, select the next highest number.

13. Click **OK**.
14. Click **Restart** on the Runtime tab to reinitialize the Runtime Performance Advisor based on the most recent saved configuration. This action also resets the state of the Runtime Performance Advisor. For example, the current warning count is reset to zero for each message.
15. Simulate a production level load. If you are using the Runtime Performance Advisor in a test environment, or doing any other tuning for performance, simulate a realistic production load for your application. The application should run this load without errors. This simulation includes numbers of concurrent users typical of peak periods, and drives system resources, such as CPU and memory to the levels expected in production. Some advice may not be accurate if sufficient load is not generated.
16. Select the check box to enable the Runtime Performance Advisor to achieve the best results for performance tuning, when a stable production level load is being applied.
17. Click **OK**.
18. Use one of these options to view tuning advice:
    - In the administrative console, select **Warnings** under the WebSphere Runtime Messages in the WebSphere Status panel.
    - View the SystemOut.log file. This file is located in the /QIBM/UserData/WebAS5/Base/*instance_name*/logs/*server_name* directory.

    Note: Some messages are not issued immediately.
19. Update the product configuration for improved performance, based on advice. Although the performance advisors attempt to distinguish between loaded and idle conditions, the advisor might provide misleading advice if it is enabled when the system is not in one of those states. This result is especially likely when running short tests. Although the advice helps in most configurations, there might be situations where the advice hinders performance. Due to these conditions, advice is not guaranteed. Therefore, test the environment with the updated configuration to verify that it functions and performs well.

You can also enable and disable advice on the **Advice Configuration** page. Some advice applies only to certain configurations, and can only be enabled for those configurations. For example, **Unbounded ORB Service Thread Pool** advice is only relevant when the ORB Service thread pool is unbounded, and can only be enabled when the ORB thread pool is unbounded. For more information about advice configuration, see Advice configuration settings.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY  10594-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

© Copyright IBM Corp. 1998, 2004

```
IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both: WebSphere
AS/400
e (logo)
IBM
iSeries
Operating System/400
OS/400
400

Lotus, Freelance, and WordPro are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Terms and conditions for downloading and printing publications

Permissions for the use of the publications you have selected for download are granted subject to the following terms and conditions and your indication of acceptance thereof.

Personal Use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these Publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

All material copyrighted by IBM Corporation.

By downloading or printing a publication from this site, you have indicated your agreement with these terms and conditions.

Code example disclaimer

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you ″AS IS″ without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

IBM®

Printed in USA