IBM WebSphere Application Server, Version 5.0.2

Monitoring and Tuning Performance

IBM

Compilation date: July 30, 2003

Contents

Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- Cloudscape
- Everyplace
- iSeries
- IBM
- Redbooks
- ViaVoice
- WebSphere
- zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. 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.

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.

Chapter 1. Welcome to monitoring and tuning performance

To help administrators identify application performance problems, the product collects performance data and provides interfaces that allow external applications to monitor the performance data, tools to display performance data for analysis, and tools to suggest performance tuning improvements.

In addition, problem determination capabilities in the product help administrators by providing APIs and graphical interfaces for collecting and analyzing trace and log data.

Performance Monitoring Infrastructure (PMI)

The product collects data on run-time and applications through the Performance Monitoring Infrastructure (PMI), as described in Performance Monitoring Infrastructure. This infrastructure is compatible with and extends the JSR-077 specification.

PMI provides several types of interfaces to access performance data. A new JMX API is introduced in this version, but the servlet and Java client interfaces are still available for compatibility with Versions 3.5.5+ and 4.0+. These PMI interfaces are used to create tools to help monitor and tune performance.

Performance data can be monitored and analyzed with:
- Tivoli Performance Viewer formerly known as Resource Analyzer, which is included in WebSphere Application Server
- Other Tivoli Monitoring Tools
- User-developed monitoring tools
- Third-party monitoring tools

The Tivoli Performance Viewer uses the PMI Java client to provide graphical displays and summary reports of collected data. For more information, see Monitoring performance with Tivoli Performance Viewer (formerly Resource Analyzer).

Performance Monitoring Infrastructure (PMI) Request Metrics

IBM WebSphere Application Server also collects data by timing requests as they travel through the product components. PMI Request Metrics logs time spent in major components, such as Web Server, Web container, Enterprise bean container, and database. These data points are recorded in logs and can be written to Application Response Time (ARM) agents used by Tivoli monitoring tools.

For more information about PMI Request Metrics, see Measuring data requests (Performance Monitoring Infrastructure Request Metrics).

5.0.2 **Performance Advisors**

5.0.2 IBM WebSphere Application Server includes two performance advisors to help tune systems for optimal performance. Both advisors use the PMI data to suggest configuration changes to Object Request Broker (ORB) service thread pools,

Web container thread pools, connection pool size, persisted session size and time, prepared statement cache size, and session cache size. The Runtime Performance Advisor runs in the application server process, while the other advisor runs in the Tivoli Performance Viewer (TPV). The following summary chart outlines the difference between the two advisors. For more information, see the articles, Using the Runtime Performance Advisor and Using the Performance Advisor in Tivoli Performance Viewer.

	Runtime Performance Advisor	Performance Advisor in Tivoli Performance Viewer (TPV)
Location of execution	Application server	TPV client
Location of tool	Administrative console	TPV
Output	SystemOut.log file and WebSphere run-time messages in WebSphere status panel in the administrative console	TPV graphical user interface (GUI)
Frequency of operation	Every 10 seconds in background	When you select refresh in TPV
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

Chapter 2. Monitoring performance

WebSphere Application Server collects data on run-time and applications through the Performance Monitoring Infrastructure (PMI). Performance data can then be monitored and analyzed with a variety of tools.

Steps for this task

1. Enable performance monitoring services in the application server through the administrative console and Enable performance monitoring services in the node agent through the administrative console if running WebSphere Application Server Network Deployment.

   In order to monitor performance data through the PMI interfaces, you must first enable the performance monitoring service through the administrative console and restart the server. If running in Network Deployment, you need to enable PMI services on both the server and on the node agent and restart the server and the node agent.

2. Enable performance monitoring services in the NodeAgent through the administrative console.

   In order to monitor performance data through the PMI interfaces, you must first enable the performance monitoring services through the administrative console and restart the server. If running Network Deployment product, you need to enable PMI service on both the server and on the node agent and restart the server and the node agent.

3. Collect the data.

   The monitoring levels that determine which data counters are enabled can be set dynamically, without restarting the server. This can be done in one of the following ways:

   a. Enable data collection through the administrative console.

   b. Enable performance monitoring services through Tivoli Performance Viewer (formerly Resource Analyzer).

   c. Enable performance monitoring services using the command line.

4. Monitor and analyze performance data.

   You can monitor and analyze data with several tools:

   a. Monitor performance data with Tivoli Performance Viewer.

      This tool is included with WebSphere Application Server.

   b. Monitor performance data with other Tivoli monitoring tools.

   c. Monitor performance data with user-developed monitoring tools.

      Write your own applications to monitor performance data.

   d. Monitor performance with third-party monitoring tools.

What to do next

WebSphere Application Server also collects data through PMI Request Metrics. This feature times requests as they travel through WebSphere Application Server components. For more information about PMI Request Metrics see the topic ″Measuring data requests (Performance Monitoring Infrastructure Request Metrics)″.

Performance Monitoring Infrastructure

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 using a Web client, Java client or JMX client. WebSphere Application Server contains Tivoli Performance Viewer, a Java client which displays and monitors performance data. See the topics Monitoring performance with Tivoli Performance Viewer (formerly Resource Analyzer), Tivoli performance monitoring and management solutions, Third-party performance monitoring and management solutions, and Developing your own monitoring applications for more information on monitoring tools".

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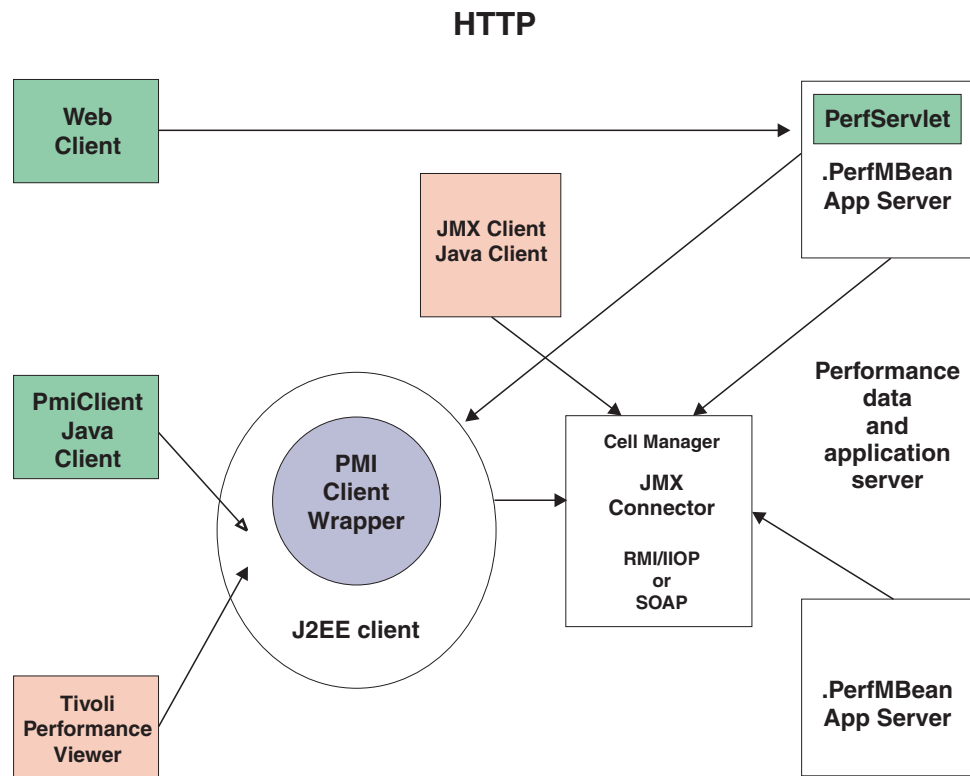
The figure shows the overall PMI architecture. On the right side, the server updates and keeps PMI data in memory. The left side displays a Web client, Java client and JMX client retrieving the performance data.

Performance data organization

Performance Monitoring Infrastructure (PMI) provides server-side monitoring and a client-side API to retrieve performance data. PMI maintains statistical data within the entire WebSphere Application Server domain, including multiple nodes and servers. Each node can contain one or more WebSphere Application Servers. Each server organizes PMI data into modules and submodules.

Hiearchy of data collections
used for performance reporting
to Resource Analyzer

Node* ↔ PMI Client ↔ **Resource Analyzer**

Counter   Serve 2   Server*

Instance

Module
Enterprise beans   Avg Method RT

EJB Module 1

entity   stateful   stateless

Bean1   Bean2   Bean3

Avg Method RT   Gets Found   Num Destroys   Num Creates   Methods   Avg Method RT
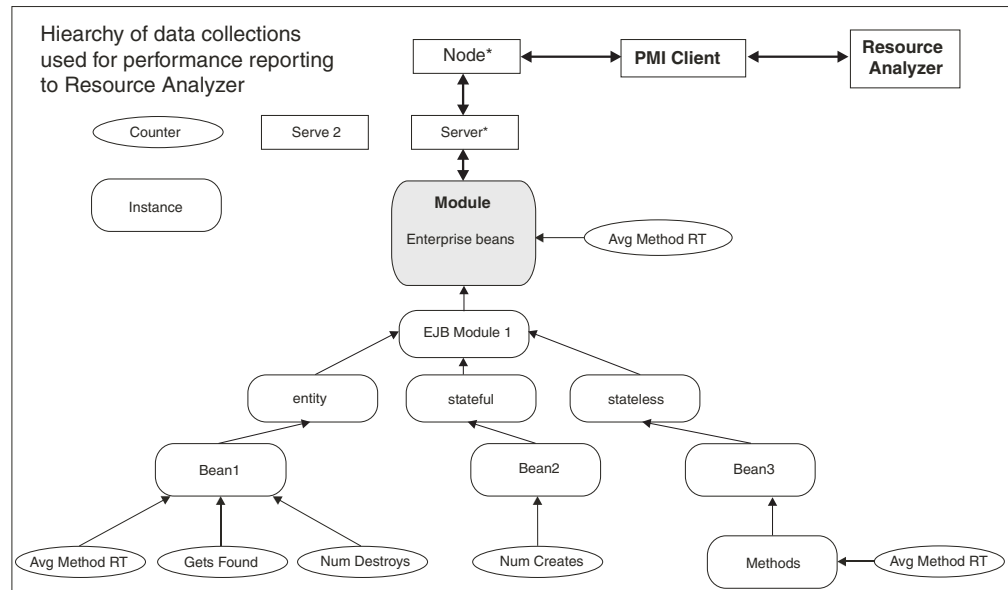
The Tivoli Performance Viewer, formerly the Resource Analyzer, organizes
performance data in a centralized hierarchy of the following objects:

- **Node.** A node represents a physical machine in the WebSphere Application
  Server administrative domain.
- **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 performance viewer. Each module has a configuration file
  in XML format. This file determines organization and lists a unique identifier for
  each performance data 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), Web Services Gateway (WSGW),
  and dynamic cache.
- **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 used to hold performance information for
  analysis. Each resource category (module) has an associated set of counters. The
  data points within a module are queried and distinguished by the Mbean
  ObjectNames or PerfDescriptors. Examples of counters include the number of
  active enterprise beans, the time spent responding to a servlet request and the
  number of kilobytes of available memory.

The Tivoli Performance Viewer allows users 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. For
example, in the figure above, both the Enterprise beans module and Bean1 have an
Avg Method RT counter.

Counters are enabled at the module level and can be enabled or disabled for
elements within the module. For example, in the figure, if the Enterprise beans

module is enabled, its Avg Method RT counter is enabled by default. However, you can then disable the Avg Method RT counter even when the rest of the module counters are enabled. You can also, if desired, disable the Avg Method RT counter for Bean1, but the aggregate response time reported for the whole module will no longer include Bean1 data.

Each counter has a specified monitoring level: none, low, medium, high or maximum. If the module is set to lower monitoring level than required by a particular counter, that counter will not be enabled. Thus, if Bean1 has a medium monitoring level, Gets Found and Num Destroys are enabled because they require a low monitoring level. However, Avg Method RT is not enabled because it requires a high monitoring level.

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.
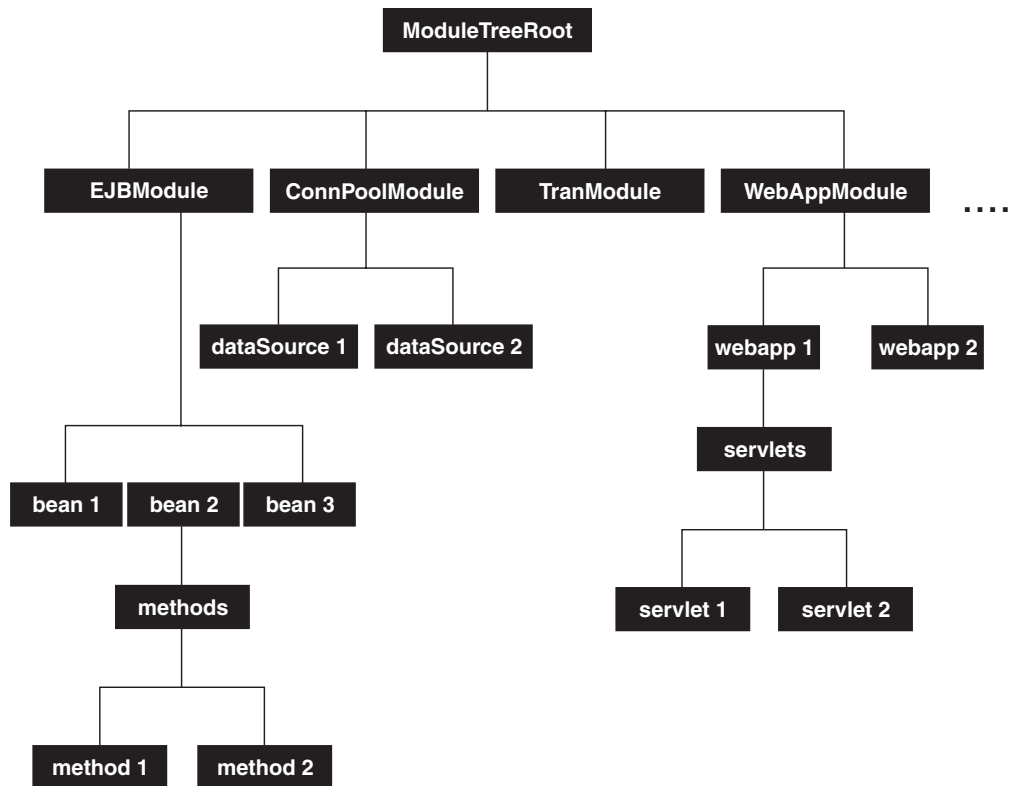
The following PMI modules are available to provide statistical data:
- Enterprise bean module, enterprise bean, methods in a bean

   Data counters for this category report load values, response times, and life cycle activities for enterprise beans. Examples include the average number of active beans and the number of times bean data is loaded or written to the database. Information is provided for enterprise bean methods and the remote interfaces used by an enterprise bean. Examples include the number of times a method is called and the average response time for the method. In addition, the Tivoli Performance Viewer reports information on the size and use of a bean objects cache or enterprise bean object pool. Examples include the number of calls attempting to retrieve an object from a pool and the number of times an object is found available in the pool.

- JDBC connection pools

  Data counters for this category contain usage information about connection pools for a database. Examples include 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, and the average time the connection is in use.

- J2C connection pool

  Data counters for this category contain usage information about the Java 2 Enterprise Edition (J2EE) Connector Architecture that enables enterprise beans to connect and interact with procedural back-end systems, such as Customer Information Control System (CICS), and Information Management System (IMS). Examples include the number of managed connections or physical connections and the total number of connections or connection handles.

- Java Virtual Machine API (JVM)

  Data counters for this category contain memory used by a process as reported by Java Virtual Machine (JVM) run time. Examples are the total memory available and the amount of free memory for the JVM. 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.

- Servlet session manager

  Data counters for this category contain usage information for HTTP sessions. Examples include the total number of accessed sessions, the average amount of time it takes for a session to perform a request, and the average number of concurrently active HTTP sessions.

- Thread pool

  Data counters for this category contain information about the thread pools for Object Request Broker (ORB) threads and the Web container pools used to process HTTP requests. Examples include the number of threads created and destroyed, the maximum number of pooled threads allowed, and the average number of active threads in the pool.

- Java Transaction API (JTA)

  Data counters for this category contain performance information for the transaction manager. Examples include the average number of active transactions, the average duration of transactions, and the average number of methods per transaction.

- Web applications, servlet

  Data counters for this category contain information for the selected server. Examples include the number of loaded servlets, the average response time for completed requests, and the number of requests for the servlet.

- Object Request Broker (ORB)

  Data counters for this category contain information for the ORB. Examples include the object reference lookup time, the total number of requests, and the processing time for each interceptor.

- Web Services Gateway (WSGW)

  Data counters for this category contain information for WSGW. Examples include the number of synchronous and asynchronous requests and responses.

- System data

  Data counters for this category contain information for a machine (node). Examples include the CPU utilization and memory usage. Note that this category is available at node level, which means it is only available at NodeAgent in the multiple servers version.

- Workload Management (WLM)

  Data counters for this category contain information for workload management. Examples include the number of requests, the number of updates and average response time.
- Dynamic cache

  Data counters for this category contain information for the dynamic cache service. Examples include in memory cache size, number of invalidations and number of hits and misses.
- Web Services

  Data counters for this category contain information for the web services. Examples include number of loaded web services, number of requests delivered and processed, request response time, and average size of requests.

BeanModule data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
creates	Number of times beans were created	3.5.5 and above	per home	CountStatistic	Low
removes	Number of times beans were removed	3.5.5 and above	per home	CountStatistic	Low
passivates	Number of times beans were passivated (entity and stateful)	3.5.5 and above	per home	CountStatistic	Low
activates	Number of times beans were activated (entity and stateful)	3.5.5 and above	per home	CountStatistic	Low
persistence loads	Number of times bean data was loaded from persistent storage (entity)	3.5.5 and above	per home	CountStatistic	Low
persistence stores	Number of times bean data was stored in persistent storage (entity)	3.5.5 and above	per home	CountStatistic	Low
instantiations	Number of times bean objects were instantiated	3.5.5 and above	per home	CountStatistic	Low
destroys	Number of times bean objects were freed	3.5.5 and above	per home	CountStatistic	Low

Num Ready Beans	Number of concurrently ready beans (entity and session). This counter was called concurrent active in Versions 3.5.5+ and 4.0.	3.5.5 and above	per home	RangeStatistic	High
concurrent live	Number of concurrently live beans	3.5.5 and above	per home	RangeStatistic	High
avg method rsp time	Average response time in milliseconds on the bean methods (home, remote, local)	3.5.5 and above	per home	TimeStatistic	High
avg method rsp time for create	Average time in milliseconds a bean create call takes including the time for the load if any	5.0	per home	TimeStatistic	Medium
avg method rsp time for load	Average time in milliseconds for loading the bean data from persistent storage (entity)	5.0	per home	TimeStatistic	Medium
avg method rsp time for store	Average time in milliseconds for storing the bean data to persistent storage (entity)	5.0	per home	TimeStatistic	Medium
avg method rsp time for remove	Average time in milliseconds a bean entries call takes including the time at the database, if any	5.0	per home	TimeStatistic	Medium
total method calls	total number of method calls	3.5.5 and above	per home	CountStatistic	High
avg method rsp time for activation	Average time in milliseconds a beanActivate call takes including the time at the database, if any	5.0	per home	TimeStatistic	Medium
avg method rsp time for passivation	Average time in milliseconds a beanPassivate call takes including the time at the database, if any	5.0	per home	TimeStatistic	Medium

active methods	Number of concurrently active methods - num methods called at the same time.	3.5.5 and above	per home	TimeStatistic	High
Per method invocations	Number of calls to the bean methods (home, remote, local)	3.5.5 and above	per method/per home	CountStatistic	Max
Per method rsp time	Average response time in milliseconds on the bean methods (home, remote, local)	3.5.5 and above	per home	TimeStatistic	Max
Per method concurrent invocations	Number of concurrent invocations to call a method	5.0	per method/per home	RangeStatistic	Max
getsFromPool	Number of calls retrieving an object from the pool (entity and stateless)	3.5.5 and above	per home/object pool	CountStatistic	Low
getsFound	Number of times a retrieve found an object available in the pool (entity and stateless)	3.5.5 and above	per home/object pool	CountStatistic	Low
returnsToPool	Number of calls returning an object to the pool (entity and stateless)	3.5.5 and above	per home/object pool	CountStatistic	Low
returnsDiscarded	Number of times the returning object was discarded because the pool was full (entity and stateless)	3.5.5 and above	per home/object pool	CountStatistic	Low
drainsFromPool	Number of times the daemon found the pool was idle and attempted to clean it (entity and stateless)	3.5.5 and above	per home/object pool	CountStatistic	Low
avgDrainSize	Average number of objects discarded in each drain (entity and stateless)	3.5.5 and above	per home/object pool	TimeStatistic	Medium
avgPoolSize	Number of objects in the pool (entity and stateless)	3.5.5 and above	per home/object pool	RangeStatistic	High

messageCount	Number of messages delivered to the bean onMessage method (message driven beans)	5.0	per type	CountStatistic	Low
messageBackoutCount	Number of messages failed to be delivered to the bean onMessage method (message driven beans)	5.0	per type	CountStatistic	Low
serverSessionWait	Average time to obtain a ServerSession from the pool (message drive bean)	5.0	per type	TimeStatistic	Medium
serverSessionUsage	Percentage of server session pool in use (message driven)	5.0	per type	RangeStatistic	High

JDBC connection pool data counters

PMI collects performance data for 4.0 and 5.0 JDBC data sources. For a 4.0 data source, the data source name is used. For a 5.0 data source, the JNDI name is used.

The JDBC connection pool counters are used to monitor the JDBC data sources performance. The data can be found by using the Tivoli Performance Viewer and looking under each application server. Click *application_server* > **JDBC connection pool**.

Data counter definitions

Name	Description	Version	Granularity	Type	Level
creates	Total number of connections created	3.5.5 and above	per connection pool	CountStatistic	Low
avg pool size	Average pool size	3.5.5 and above	per connection pool	BoundedRangeStatistic	High
free pool size	Average free pool size	5.0	per connection pool	BoundedRangeStatistic	High
allocates	Total number of connections allocated	3.5.5 and above	per connection pool	CountStatistic	Low
returns	Total number of connections returned	4.0 and above	per connection pool	CountStatistic	Low
avg waiting threads	Number of threads that are currently waiting for a connection	3.5.5 and above	per connection pool	RangeStatistic	High
connection pool faults	Total number of faults, such as, timeouts, in connection pool	3.5.5 and above	per connection pool	CountStatistic	Low

destroys	Number of times bean objects were freed	3.5.5 and above	per connection pool	CountStatistic	Low
avg wait time	Average waiting time in milliseconds until a connection is granted	5.0	per connection pool	TimeStatistic	Medium
avg time in use	Average time a connection is used (Difference between the time at which the connection is allocated and returned. This includes the JDBC operation time.)	5.0	per connection pool	TimeStatistic	Medium
percent used	Average percent of the pool that is in use	3.5.5 and above	per connection pool	RangeStatistic	High
percent maxed	Average percent of the time that all connections are in use	3.5.5 and above	per connection pool	RangeStatistic	High
Statement cache discard count	Total number of statements discarded by the LRU algorithm of the statement cache	4.0 and above	per connection pool	CountStatistic	Low
Number managed connections	Number of ManagedConnection objects in use	5.0	per connection factory	CountStatistic	Low
Number connections	Current number of connection objects in use	5.0	per connection factory	CountStatistic	Low
jdbcOperationTimer	Amount of time in milliseconds spent executing in the JDBC driver (includes time spent in JDBC driver, network and database)	5.0	per data source	TimeStatistic	Medium

J2C connection pool data counters

The J2C connection pool data counters are used to monitor the J2C connection pool performance. The data can be found by using the Tivoli Performance Viewer and looking under each application server. Click *application_server* > **J2C connection pool**.

Data counter definitions

Name	Description	Version	Granularity	Type	Level
Number managed connections	Number of ManagedConnection objects in use	5.0	per connection factory	CountStatistic	Low

Number connections	Current number of connection objects in use	5.0	per connection factory	CountStatistic	Low
Number managed connections created	Total number of connections created	5.0	per connection factory	CountStatistic	Low
Number managed connections destroyed	Total number of connections destroyed	5.0	per connection factory	CountStatistic	Low
Number managed connections allocated	Total number of connections allocated	5.0	per connection factory	CountStatistic	Low
Number managed connections freed	Total number of connections freed	5.0	per connection factory	CountStatistic	Low
faults	Number of faults, such as timeouts, in connection pool	5.0	per connection factory	CountStatistic	Low
free pool size	Number of free connections in the pool	5.0	per connection factory	BoundedRangeStatistic	High
pool size	Pool size	5.0	per connection factory	BoundedRangeStatistic	High
concurrent waiters	Average number of threads concurrently waiting for a connection	5.0	per connection factory	RangeStatistic	High
Percent used	Average percent of the pool that is in use	5.0	per connection factory	RangeStatistic	High
Percent maxed	Average percent of the time that all connections are in use	5.0	per connection factory	RangeStatistic	High
Average wait time	Average waiting time in milliseconds until a connection is granted	5.0	per connection factory	TimeStatistic	Medium
Average use time	Average time in milliseconds that connections are in use	5.0	per connection factory	TimeStatistic	Medium

Java Virtual Machine data counters

The Java Virtual Machine (JVM) data counters are used to monitor the JVM performance. With an exception to the data counters used for total, used and free heap size, the data counters can be found using the Java Virtual Machine Profiler Interface (JVMPI). In order to use JVMPI, you must turn on the monitoring by settting the **-XrunpmiJvmpiProfiler** command line. See Enabling Java Virtual Machine Profiler Interface data reporting

Data counter definitions

Name	Description	Version	Granularity	Type	Level
Free memory	Free memory in JVM run time	3.5.5 and above	per Java Virtual Machine (JVM)	CountStatistic	Low
Used memory	Used memory in JVM run time	3.5.5 and above	per JVM	CountStatistic	Low

Total memory	Total memory in JVM run time	3.5.5 and above	per JVM	BoundedRangeStatistic. The upperBound and lowerBound are not implemented for the Total memory counter.	High
Up time	The amount of time the JVM is running	5.0	per JVM	CountStatistic	Low
Number garbage collection calls	Number of garbage collection calls. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	CountStatistic	Max
Average time between garbage collection	Average garbage collection in seconds between two garbage collection. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	TimeStatistic	Max
Average garbage collection duration	Average duration of a garbage collection. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	TimeStatistic	Max
num waits for a lock	Number of times that a thread waits for a lock.This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	CountStatistic	Max
avg time waiting for lock	Average time that a thread waits for a lock. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	TimeStatistic	Max
Number of objects allocated	Number of objects allocated in heap. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	CountStatistic	Max
Number of objects found	Number of objects in heap. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM.	4.0 and above	per JVM	CountStatistic	Max

| Number of objects freed | Number of objects freed in heap. This counter is not available unless `-XrunpmiJvmpiProfiler` is set when starting the JVM. | 4.0 and above | per JVM | CountStatistic | Max |

Object Request Broker data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
referenceLookupTime	The time (in milliseconds) to look up an object reference before method dispatch can be carried out	5.0	Object Request Broker (ORB)	TimeStatistic	Medium
numRequest	The total number of requests sent to the ORB	5.0	ORB	CountStatistic	Low
concurrentRequests	The number of requests that are concurrently processed by the ORB	5.0	ORB	RangeStatistic	High
processingTime	The time (in milliseconds) it takes a registered portable interceptor to run	5.0	per interceptor	TimeStatistic	Medium

Session data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
createdSessions	Number of sessions created	3.5.5 and above	per web application	CountStatistic	Low
invalidatedSessions	Number of sessions invalidated	3.5.5 and above	per web application	CountStatistic	Low
sessionLifeTime	The average session lifetime	3.5.5 and above	per web application	TimeStatistic	Medium
activeSessions	The number of concurrently active sessions. A session is active if WebSphere is currently processing a request which uses that session.	3.5.5 and above	per web application	RangeStatistic	High
liveSession	The number of sessions that are currently cached in memory	5.0 and above	per web application	RangeStatistic	High

NoRoomForNewSession	Applies only to session in memory with AllowOverflow=false. The number of times that a request for a new session can not be handled because it would exceed the maximum session count.	5.0	per Web application	CountStatistic	Low
cacheDiscards	Number of session objects that have been forced out of the cache. (An LRU algorithm removes old entries to make room for new sessions and cache misses). Applicable only for persistent sessions.	5.0	per Web application	CountStatistic	Low
externalReadTime	Time (milliseconds) taken in reading the session data from persistent store. For multirow sessions, the metrics are for the attribute; for single row sessions, the metrics are for the whole session. Applicable only for persistent sessions. When using a JMS persistent store, the user has the choice of whether to serialize the data being replicated. If they choose not to serialize the data, the counter will not be available.	5.0	per Web application	TimeStatistic	Medium
externalReadSize	Size of session data read from persistent store. Applicable only for (serialized) persistent sessions; similar to externalReadTime above.	5.0	per Web application	TimeStatistic	Medium

externalWriteTime	Time (milliseconds) taken to write the session data from the persistent store. Applicable only for (serialized) persistent sessions. Similar to externalReadTime above.	5.0	per Web application	TimeStatistic	Medium
externalWriteSize	Size of session data written to persistent store. Applicable only for (serialized) persistent sessions. Similar to externalReadTime above.	5.0	per Web application	TimeStatistic	Medium
affinityBreaks	The number of requests received for sessions that were last accessed from another Web application. This can indicate failover processing or a corrupt plug-in configuration.	5.0	per Web application	CountStatistic	Low
serializableSessObjSize	The size in bytes of (the serializable attributes of) in-memory sessions. Only count session objects that contain at least one serializable attribute object. Note that a session may contain some attributes that are serializable and some that are not. The size in bytes is at a session level.	5.0	per Web application	TimeStatistic	Max
timeSinceLastActivated	The time difference in milliseconds between previous and current access time stamps. Does not include session time out.	5.0	per Web application	TimeStatistic	Medium
invalidatedViaTimeout	The number of requests for a session that no CountStatistic exists, presumeably because the session timed out.	5.0	per Web application	CountStatistic	Low

attemptToActivateNotExistentSession	Number of requests for a session that no longer exists, presumeably because the session timed out. Use this counter to help determine if the timeout is too short.	5.0	per Web application	CountStatistic	Low

Transaction data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
Number global transactions begun	Total number of global transactions begun on server	4.0 and above	per transaction manager/server	CountStatistic	Low
Number global transactions involved	Total number of global trans involved on server (for example, begun and imported)	4.0 and above	per transaction manager/server	CountStatistic	Low
Number local transactions begun	Total number of local transactions begun on server	4.0 and above	per transaction manager/server	CountStatistic	Low
Active global transactions	Number of concurrently active global transactions	3.5.5 and above	per transaction manager/server	CountStatistic	Low
Active local transactions	Number of concurrently active local transactions	4.0 and above	per transaction manager/server	CountStatistic	Low
Global transactions duration	Average duration of global transactions	3.5.5 and above	per transaction manager/server	TimeStatistic	Medium
Local transaction duration	Average duration of local transactions	4.0 and above	per transaction manager/server	TimeStatistic	Medium
Local transactions before_completion time	Average duration of before_completion for local transactions	4.0 and above	per transaction manager or server	TimeStatistic	Medium
Global transaction commit time	Average duration of commit for global transactions	4.0 and above	per transaction manager/server	TimeStatistic	Medium

Global transaction prepare time	Average duration of prepare for global transactions	4.0 and above	per transaction manager/server	TimeStatistic	Medium
Local transaction before_completion time	Average duration of before_completion for local transactions	4.0 and above	per transaction manager/server	TimeStatistic	Medium
Local transaction commit time	Average duration of commit for local transactions	4.0 and above	per transaction manager/server	TimeStatistic	Medium
Number global transactions committed	Total number of global transactions committed	3.5.5 and above	per transaction manager/server	CountStatistic	Low
Number of global transactions rolled back	Total number of global transactions rolled back	3.5.5 and above	per transaction manager/server	CountStatistic	Low
Number global transactions optimized	Number of global transactions converted to single phase for optimization	4.0 and above	per transaction manager/server	CountStatistic	Low
Number of local transactions committed	Number of local transactions committed	4.0 and above	per transaction manager/server	CountStatistic	Low
Number of local transactions rolled back	Number of local transactions rolled back	4.0 and above	per transaction manager/server	CountStatistic	Low
Number of global transactions timed out	Number of global transactions timed out	4.0 and above	per transaction manager/server	CountStatistic	Low
Number of local transactions timed out	Number of local transactions timed out	4.0 and above	per transaction manager/server	CountStatistic	Low

ThreadPool data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
Thread creates	Total number of threads created	3.5.5 and above	per thread pool	CountStatistic	Low
Thread destroys	Total number of threads destroyed	3.5.5 and above	per thread pool	CountStatistic	Low
Active threads	The number of concurrently active threads	3.5.5 and above	per thread pool	RangeStatistic	High
Pool size	Average number of threads in pool	3.5.5 and above	per thread pool	BoundedRangeStatistic	High

Percent maxed	Average percent of the time that all threads are in use	3.5.5 and above	per thread pool	RangeStatistic	High

Web application data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
numLoadedServlets	Number of servlets that were loaded	3.5.5 and above	per Web application	CountStatistic	Low
numReloads	Number of servlets that were reloaded	3.5.5 and above	per Web application	CountStatistic	Low
totalRequests	Total number of requests a servlet processed	3.5.5 and above	per servlet	CountStatistic	Low
concurrentRequests	Number of requests that are concurrently processed	3.5.5 and above	per servlet	RangeStatistic	High
responseTime	The response time, in milliseconds, of a servlet request	3.5.5 and above	per servlet	TimeStatistic	Medium
numErrors	Total number of errors in a servlet or Java Server Page (JSP)	3.5.5 and above	per servlet	CountStatistic	Low

Workload Management data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
numIncomingRequests	Total number of incoming IIOP requests to an application server	5.0	per server	CountStatistic	Low

numIncomingStrongAffinityRequests	Number of incoming IIOP requests to an application server that are based on a strong affinity. A strong affinity request is defined as a request that must be serviced by this application server because of a dependency that resides on the server. This request could not successfully be serviced on another member in the server cluster. In Version 5.0 ND edition, transactional affinity is the only example of a strong affinity	5.0	per server	CountStatistic	Low
numIncomingNonAffinityRequests	Number of incoming IIOP requests to an application server based on no affinity. This request was sent to this server based on workload management selection policies that were decided in the Workload Management (WLM) run time of the client.	5.0	per server	CountStatistic	Low
numIncomingNonWLMObjectRequests	Number of incoming IIOP requests to an application server that came from a client that does not have the WLM run time present or where the object reference on the client was flagged not to participate in workload management.	5.0	per server	CountStatistic	Low

numServerClusterUpdates	Number of times initial or updated server cluster data is sent to a server member from the deployment manager. This metric determines how often cluster information is being propagated.	5.0	per server	CountStatistic	Low
numOfWLMClientServiced	Number of WLM enabled clients that have been serviced by this application server.	5.0	per server	CountStatistic	Low
numOfConcurrentRequests	Number of remote IIOP requests currently being processed by this server	5.0	per server	RangeStatistic	High
serverResponseTime	The response time (in milliseconds) of IIOP requests being serviced by an application server. The response time is calculated based on the time the request is received to the time when the reply is sent back out.	5.0	per server	TimeStatistic	Medium
numOfOutgoingRequests	The total number of outgoing IIOP requests being sent from a client to an application server	5.0	per WLM	CountStatistic	Low
numClientClusterUpdates	The number of times initial or updated server cluster data is sent to a WLM enabled client from server cluster member. Use this metric to determine how often cluster information is being propagated.	5.0	per WLM	CountStatistic	Low

clientResponseTime	The response time (in milliseconds) of IIOP requests being sent from a client. The response time is calculated based on the time the request is sent from the client to the time the reply is received from the server.	5.0	per WLM	TimeStatistic	Medium

System data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
percentCpuUsage	The average system CPU utilization taken over the time interval since the last reading. Because the first call is required to perform initialization, an invalid value such as 0 will be returned. All subsequent calls will return the expected value. On SMP machines, the value returned will be the utilization averaged over all CPUs.	5.0	per node	CountStatistic	Low
freeMemory	The amount of real free memory available on the system. Real memory that is not allocated is only a lower bound on available real memory, since many operating systems take some of the otherwise unallocated memory and use it for additional I/O buffering. The exact amount of buffer memory which can be freed up is dependent on both the platform and the application(s) running on it.	5.0	per node	CountStatistic	Low
avgCpuUsage	The average percentCpuUsage that is busy after the server is started	5.0	per node	TimeStatistic	Medium

Dynamic cache data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
maxInMemoryCacheSize	Maximum number of in-memory cache entries	5.0	per server	CountStatistic	Low
inMemoryCacheSize	Current number of in-memory cache entries	5.0	per server	CountStatistic	Low
totalTimeoutInvalidation	Aggregate of template timeouts and disk timeouts	5.0	per server	CountStatistic	Low
hitsInMemory	Requests for this cacheable object served from memory	5.0	per template	CountStatistic	Low
hitsOnDisk	Requests for this cacheable object served from disk	5.0	per template	CountStatistic	Low
explicitInvalidations	Total explicit invalidation issued for this template	5.0	per template	CountStatistic	Low
lruInvalidations	Cache entries evicted from memory by a Least Recently Used algorithm. These entries are passivated to disk if disk overflow is enabled.	5.0	per template	CountStatistic	Low
timeoutInvalidations	Cache entries evicted from memory and/or disk because their timeout has expired	5.0	per template	CountStatistic	Low
Entries	Current number of cache entries created from this template. Refers to the per-template equivalent of totalCacheSize.	5.0	per template	CountStatistic	Low
hitsRemove	Requests for this cacheable object served from other Java Virtual Machines (JVM) in the cluster	5.0	per template	CountStatistic	Low
Misses	Requests for this cacheable object that were not found in the cache	5.0	per template	CountStatistic	Low

RequestFromClient	Requests for this cacheable object generated by applications running on the application server	5.0	per template	CountStatistic	Low
requestsFromJVM	Requests for this cacheable object generated by cooperating caches in this cluster	5.0	per template	CountStatistic	Low
explicitInvalidationsFromMemory	Explicit invalidations resulting in an entry being removed from memory	5.0	per template	CountStatistic	Low
explicitInvalidationsFromDisk	Explicit invalidations resulting in an entry being removed from disk	5.0	per template	CountStatistic	Low
explicitInvalidationsNoOp	Explicit invalidations received for this template where no corresponding entry exists	5.0	per template	CountStatistic	Low
explicitInvalidationsLocal	Explicit invalidations generated locally, either programmatically or by a cache policy	5.0	per template	CountStatistic	Low
explicitInvalidationsRemote	Explicit invalidations received from a cooperating JVM in this cluster	5.0	per template	CountStatistic	Low
remoteCreations	Entries received from cooperating dynamic caches	5.0	per template	CountStatistic	Low

Web Services Gateway (WSGW) data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
synchronousRequests	Number of synchronous requests that were made	5.0	per Web service	CountStatistic	Low
synchronousResponses	Number of synchronous responses that were made	5.0	per Web service	CountStatistic	Low
asynchronousRequests	Number of asynchronous requests that were made	5.0	per Web service	CountStatistic	Low

| asynchronousResponses | Number of asynchronous responses that were made | 5.0 | per Web service | CountStatistic | Low |

Web services data counters

Data counter definitions

Name	Description	Version	Granularity	Type	Level
numLoadedServices	Number of loaded Web services	5.02 and above	per service	CountStatistic	Low
numberReceived	Number of requests service received	5.02 and above	per Web service	CountStatistic	Low
numberDispatched	Number of requests service dispatched/delivered	5.02 and above	per web service	CountStatistic	Low
numberSuccessful	Number of requests service successfully processed	5.02 and above	per web service	TimeStatistic	Low
responseTime	The average response time, in milliseconds, for a successful request	5.02 and above	per web service	TimeStatistic	Medium
requestResponseTime	The average response time, in milliseconds, to prepare a request for dispatch	5.02 and above	per web service	TimeStatistic	Medium
dispatchResponseTime	The average response time, in milliseconds, to dispatch a request	5.02 and above	per web service	TimeStatistic	Medium
replyResponseTime	The average response time, in milliseconds, to prepare a reply after dispatch	5.02 and above	per web service	TimeStatistic	Medium
size	The average payload size in bytes of a received request/reply	5.02 and above	per web service	TimeStatistic	Medium
requestSize	The average payload size in bytes of a request	5.02 and above	per web service	TimeStatistic	Medium
replySize	The average payload size in bytes of a reply	5.02 and above	per web service	TimeStatistic	Medium

Performance data classification

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

The static component consists of a name, ID and other descriptive attributes to 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 the following statistical types defined in the JSR-077 specification:

- CountStatistic
- BoundaryStatistic
- RangeStatistic
- TimeStatistic
- BoundedRangeStatistic

RangeStatistic data contains current value, as well as lowWaterMark and highWaterMark.

In general, CountStatistic data require a *low* monitoring level and TimeStatistic data require 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.
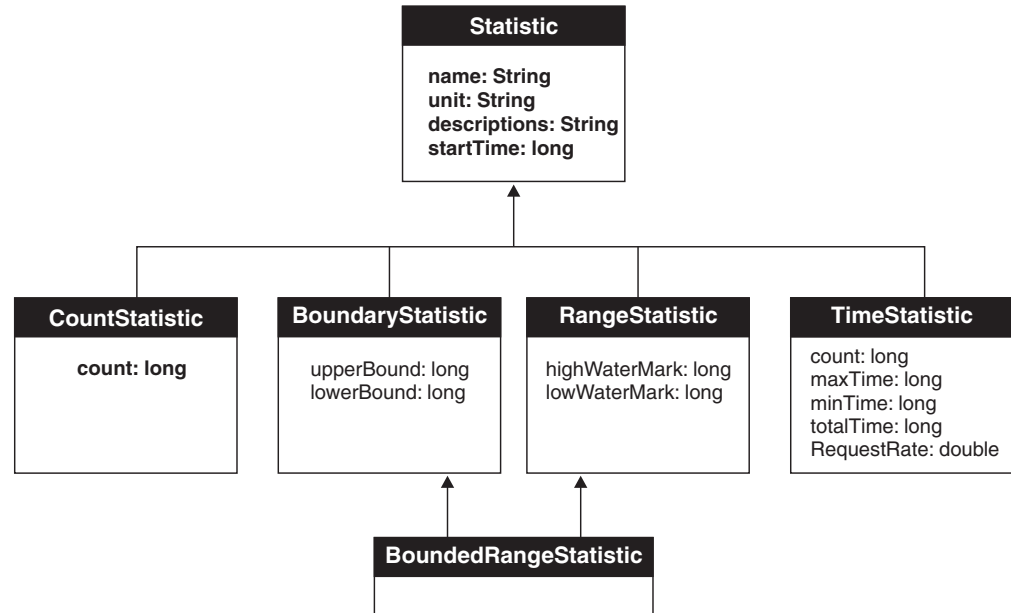
In previous versions, PMI data was classified with the following types:

- **Numeric**: Maps to CountStatistic in the JSR-077 specification. Holds a single numeric value that can either be a long or a double. This data type is used to keep track of simple numeric data, such as counts.
- **Stat**: Holds 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**: Maps to the RangeStatistic or BoundedRangeStatistic, based on JSR-077 specification. This data type keeps track of 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 keeps tracking many counter samples and then returns the total, count and average of the samples. An example of this is an average method response time. Given the nature of this statistic type, it is also used to track non-time related counters, like average read and write size. You can always call getUnit method on the data configuration information to learn the unit for the counter.

In order to reduce the monitoring overhead, numeric and stat data are not synchronized. Since these data track the total and average, the extra accuracy is generally not worth the performance cost. 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 are also synchronized to guarantee accurate values.

Enabling performance monitoring services in the application server through the administrative console

To monitor performance data through the performance monitoring infrastructure (PMI) interfaces, you must first enable PMI services through the administrative console.

Steps for this task

1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the console navigation tree.
3. Click *server*.
4. Click the **Configuration** tab.

   When in the Configuration tab, settings will apply once the server is restarted. When in the Runtime Tab, settings will apply immediately. Note that enablement of Performance Monitoring Service can only be done in the Configuration tab.
5. Click **Performance Monitoring Service**.
6. Select the checkbox **Startup**.
7. (Optional) Select the PMI modules and levels to set the **initial specification level** field.
8. Click **Apply** or **OK**.
9. Click **Save**.
10. Restart the application server.

    The changes you make will not take affect until you restart the application server.

What to do next

When running in WebSphere Application Server Network Deployment, be sure to Enable performance monitoring services in the NodeAgent through the administrative console.

Enabling performance monitoring services in the NodeAgent through the administrative console

To monitor performance data through the performance monitoring infrastructure (PMI) interfaces, you must first enable PMI services through the administrative console.

Steps for this task
1. Open the administrative console.
2. Click **System Administration** > **NodeAgents** in the console navigation tree.
3. Click *node_agent*.
4. Click **Performance Monitoring Service**.
5. Select the checkbox **Startup**.
6. (Optional) Select the PMI modules and levels to set the **initial specification level** field.
7. Click **Apply** or **OK**.
8. Click **Save**.
9. Restart the NodeAgent.

   The changes you make will not take affect until you restart the NodeAgent.

What to do next

When in the Configuration tab, settings will apply once the server is restarted. When in the Runtime Tab, settings will apply immediately. Note that enablement of Performance Monitoring Service can only be done in the Configuration tab.

Enabling performance data collection through 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.

Steps for this task
1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the console navigation tree.
3. Click *server*.
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. Click **Save**.

What to do next

These changes will take effect immediately, but will not be persistent. Use the **Configuration** tab for a persistent change. See the InfoCenter article Enabling PMI services through the administrative console for more information about making a persistent change.

Performance monitoring service settings

Use this page to specify settings for performance monitoring, including enabling performance monitoring, selecting the PMI module and setting monitoring levels.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Performance Monitoring**.

Startup

Specifies whether the application server attempts to start the specified service. If an application server is started when the performance monitoring service is disabled, you will have to restart the server in order to enable it.

Initial specification level

Specifies a Performance Monitoring Infrastructure (PMI) string that stores PMI specification levels, for example module levels, for all components in the server.

Set the PMI specification levels by selecting the *none*, *standard* or *custom* checkbox. If you choose *none*, all PMI modules are set to the *none* level. Choosing *standard*, sets all PMI modules to *high* and enables all PMI data excluding the method level data and JVMPI data. Choosing custom, gives you the option to change the level for each individual PMI module. You can set the level to N, L, M, H or X (none, low, medium, high and maximum). **Note** that you should not change the module names.

Specifications

Specifies the PMI module and monitoring level that you have set.

Set the PMI specification levels by selecting the *none*, *standard* or *custom* checkbox. If you choose *none*, all PMI modules are set to the *none* level. Choosing *standard*, sets all PMI modules to *high* and enables all PMI data excluding the method level data and JVMPI data. Choosing custom, gives you the option to change the level for each individual PMI module. You can set the level to N, L, M, H or X (none, low, medium, high and maximum). **Note** that you should not change the module names.

Enabling performance monitoring services using the command line

You can use the command line to enable performance monitoring services.

Steps for this task
1. Enable PMI services through the administrative console.

   Make sure to restart the application server.
2. (Run the wsadmin command).

   Using **wsadmin**, you can invoke operations on Perf Mbean to obtain the PMI data, set or obtain PMI monitoring levels and enable data counters.

   The following operations in Perf MBean can be used in **wsadmin**:

   ```
   /** Set instrumentation level using String format
   *   This should be used by scripting for an easy String processing
   */ The level STR is a list of moduleName=Level connected by ":".
   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 will 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

Usage scenario

The following are sample commands in **wsadmin** you can use to obtain PMI data:

Obtain the Perf MBean ObjectName
```
wsadmin>set perfName [$AdminControl completeObjectName type=Perf,*]
wsadmin>set perfOName [$AdminControl makeObjectName $perfName]
```

Invoke getInstrumentationLevelString operation
- use invoke since it has no parameter

  ```
  wsadmin>$AdminControl invoke $perfName getInstrumentationLevelString
  ```

This command returns the following:

```
beanModule=H:cacheModule=H:connectionPoolModule=H:j2cModule=H:jvmRuntimeModule=H
:orbPerfModule=H:servletSessionsModule=H:systemModule=H:threadPoolModule=H
:trans actionModule=H:webAppModule=H
```

Note that you can change the level (n, l, m, h, x) in the above string and then pass it to setInstrumentationLevel method.

Invoke setInstrumentationLevel operation

- set parameters ("pmi=l" is the simple way to set all modules to the low level)

```
wsadmin>set params [java::new {java.lang.Object[]} 2]
wsadmin>$params set 0 [java::new java.lang.String pmi=l]
wsadmin>$params set 1 [java::new java.lang.Boolean true]
```

- set signatures

```
wsadmin>set sigs  [java::new {java.lang.String[]} 2]
wsadmin>$sigs set 0 java.lang.String
wsadmin>$sigs set 1 java.lang.Boolean
```

- invoke the method: use invoke_jmx since it has parameter

```
wsadmin>$AdminControl invoke_jmx $perfOName setInstrumentationLevel $params $sigs
```

This command does not return anything.

Note that the PMI level string can be as simple as *pmi=level* (where level is n, l, m, h, or x) or something like *module1=level1:module2=level2:module3=level3* with the same format shown in the string returned from getInstrumentationLevelString.

Invoke getStatsString(ObjectName, Boolean) operation As an example, JVM MBean is used here.

- get MBean query string - e.g., JVM MBean

```
wsadmin>set jvmName [$AdminControl completeObjectName type=JVM,*]
```

- set parameters

```
wsadmin>set params [java::new {java.lang.Object[]} 2]
wsadmin>$params set 0 [$AdminControl makeObjectName $jvmName]
wsadmin>$params set 1 [java::new java.lang.Boolean true]
```

- set signatures

```
wsadmin>set sigs  [java::new {java.lang.String[]} 2]
wsadmin>$sigs set 0 javax.management.ObjectName wsadmin>$sigs set 1 java.lang.Boolean
```

- invoke method

```
wsadmin>$AdminControl invoke_jmx $perfOName getStatsString $params $sigs
```

This command returns the following:

```
{Description jvmRuntimeModule.desc} {Descriptor {{Node wenjianpc} {Server server
1} {Module jvmRuntimeModule} {Name jvmRuntimeModule} {Type MODULE}}} {Level 7} {
Data {{{Id 4} {Descriptor {{Node wenjianpc} {Server server1} {Module jvmRuntimeM
odule} {Name jvmRuntimeModule} {Type DATA}}} {PmiDataInfo {{Name jvmRuntimeModul
e.upTime} {Id 4} {Description jvmRuntimeModule.upTime.desc} {Level 1} {Comment {
The amount of time in seconds the JVM has been running}} {SubmoduleName null} {T
ype 2} {Unit unit.second} {Resettable false}}} {Time 1033670422282} {Value {Coun
t 638} }} {{Id 3} {Descriptor {{Node wenjianpc} {Server server1} {Module jvmRunt
imeModule} {Name jvmRuntimeModule} {Type DATA}}} {PmiDataInfo {{Name jvmRuntimeM
odule.usedMemory} {Id 3} {Description jvmRuntimeModule.usedMemory.desc} {Level 1
} {Comment {Used memory in JVM runtime}} {SubmoduleName null} {Type 2} {Unit uni
t.kbyte} {Resettable false}}} {Time 1033670422282} {Value {Count 66239} }} {{Id
2} {Descriptor {{Node wenjianpc} {Server server1} {Module jvmRuntimeModule} {Nam
e jvmRuntimeModule} {Type DATA}}} {PmiDataInfo {{Name jvmRuntimeModule.freeMemor
y} {Id 2} {Description jvmRuntimeModule.freeMemory.desc} {Level 1} {Comment {Fre
e memory in JVM runtime}} {SubmoduleName null} {Type 2} {Unit unit.kbyte} {Reset
```

```
table false}}} {Time 1033670422282} {Value {Count 34356} }} {{Id 1} {Descriptor
{{Node wenjianpc} {Server server1} {Module jvmRuntimeModule} {Name jvmRuntimeMod
ule} {Type DATA}}} {PmiDataInfo {{Name jvmRuntimeModule.totalMemory} {Id 1} {Des
cription jvmRuntimeModule.totalMemory.desc} {Level 7} {Comment {Total memory in
JVM runtime}} {SubmoduleName null} {Type 5} {Unit unit.kbyte} {Resettable false}
}} {Time 1033670422282} {Value {Current 100596} {LowWaterMark 38140} {HighWaterM
ark 100596} {MBean 38140.0} }}}}
```

Invoke getStatsString (ObjectName, String, Boolean) operation

- get MBean query string - e.g., server MBean

  ```
  wsadmin>set mySrvName [$AdminControl completeObjectName type=Server,name=server1,
  node=wenjianpc,*]
  ```

- set parameters

  ```
  wsadmin>set params [java::new {java.lang.Object[]} 3]
  wsadmin>$params set 0 [$AdminControl makeObjectName $mySrvName]
  wsadmin>$params set 1 [java::new java.lang.String beanModule]
  wsadmin>$params set 2 [java::new java.lang.Boolean true]
  ```

- set signatures

  ```
  wsadmin>set sigs  [java::new {java.lang.String[]} 3]
  wsadmin>$sigs set 0 javax.management.ObjectName
  wsadmin>$sigs set 1 java.lang.String
  wsadmin>$sigs set 2 java.lang.Boolean
  ```

- invoke method

  ```
  wsadmin>$AdminControl invoke_jmx $perfOName getStatsString $params $sigs
  ```

This command returns all the beans within the BeanModule hierarchy.

Note that this method is used to get stats data for the PMI modules that do not
have direct MBean mappings.

Invoke listStatMemberNames operation

- get MBean queryString - for example, Server

  ```
  wsadmin>set mySrvName [$AdminControl completeObjectName type=Server,name=server1,
  node=wenjianpc,*]
  ```

- set parameter

  ```
  wsadmin>set params [java::new {java.lang.Object[]} 1]
  wsadmin>$params set 0 [$AdminControl makeObjectName $mySrvName]
  ```

- set signatures

  ```
  wsadmin>set sigs  [java::new {java.lang.String[]} 1]
  wsadmin>$sigs set 0 javax.management.ObjectName
  wsadmin>$AdminControlinvoke_jmx $perfOName listStatMemberNames $params $sigs
  ```

This command returns the PMI module and submodule names, which have no
direct MBean mapping. The names are seperated by a space " ". You can then use
the name as the String parameter in getStatsString method, for example:

```
beanModule connectionPoolModule j2cModule servletSessionsModule threadPoolModule
webAppModule
```

Enabling Java Virtual Machine Profiler Interface data reporting

To enable Java Virtual Machine Profiler Interface (JVMPI) data reporting for each
individual application server:

Steps for this task

1. Open the administrative console.

2. Click **Servers** > **Application Servers** in the console navigation tree.
3. Click the application server for which JVMPI needs to be enabled.
4. Click **Process Definition**
5. Click the **Java Virtual Machine**.
6. Type -*XrunpmiJvmpiProfiler* in the **genericJvmArguments** field.
7. Click **Apply** or **OK**.
8. Click **Save**.
9. Click **Servers** > **Application Servers** in the console navigation tree.
10. Click the application server for which JVMPI needs to be enabled.
11. Click the **Configuration** tab.

    When in the Configuration tab, settings will apply once the server is restarted. When in the Runtime Tab, settings will apply immediately. Note that Performance Monitoring Service can only be enabled in the Configuration tab.
12. Click **Performance Monitoring Service**.
13. Select the checkbox **Startup**.
14. Set initial specification level to **Custom** and *jvmRuntimeModule=X*.
15. Click **Apply** or **OK**.
16. Click **Save**.
17. Start the application server, or restart the application server if it is currently running.
18. Refresh the Tivoli Performance Viewer if you are using it.

    The changes you make will not take affect until you restart the application server.

Java Virtual Machine Profiler Interface

The Tivoli Performance Viewer leverages 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, and the Java virtual machine (JVM) API 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 by enabling the user to choose which types of profiling information to collect and to select certain subsets of the time during which the JVM API is active. JVMPI moderately increases the performance impact.

Monitoring performance with Tivoli Performance Viewer (formerly Resource Analyzer)

The Resource Analyzer has been renamed *Tivoli Performance Viewer*.

Tivoli Performance Viewer is a Graphical User Interface (GUI) performance monitor for WebSphere Application Server.

Monitor and analyze the data with Tivoli Performance Viewer with these tasks:

Steps for this task

1. Start the Tivoli Performance Viewer.
2. Set monitoring levels.
3. View summary reports.
4. (Optional) Store data to a log file.
5. (Optional) Replay a performance data log file.
6. (Optional) View and modify performance chart data.
7. (Optional) Scale the performance data chart display.
8. (Optional) Refresh data.
9. (Optional) Clear values from tables and charts.
10. (Optional) Reset counters to zero.

What to do next

The Performance Advisor in Tivoli Performance Viewer provides advice to help tune systems for optimal performance and gives recommendations on inefficient settings by using collected PMI data. For more information, see Using the Performance Advisor in Tivoli Performance Viewer.

Tivoli Performance Viewer features

Tivoli Performance Viewer is a Java client which retrieves the Performance Monitoring Infrastructure (PMI) data from an application server and displays it in a variety of formats.

You can do the following tasks with the Tivoli Performance Viewer:
- View data in real time
- Record current data in a log, and replay the log later
- View data in chart form, allowing visual comparison of multiple counters. Each counter can be scaled independently to enable meaningful graphs.
- View data in tabular form
- Compare data for single resources to aggregate data across a node

To minimize the performance impact, Tivoli Performance Viewer polls the server at an interval set by the user. All data manipulations are done in the Tivoli Performance Viewer client, which can be run on a separate machine, further reducing the impact.

The Tivoli Performance Viewer graphical user interface includes the following:
- Resource selection panel
- Data monitoring panel
- Menu bar
- Toolbar icons
- Node icons
- Status bar
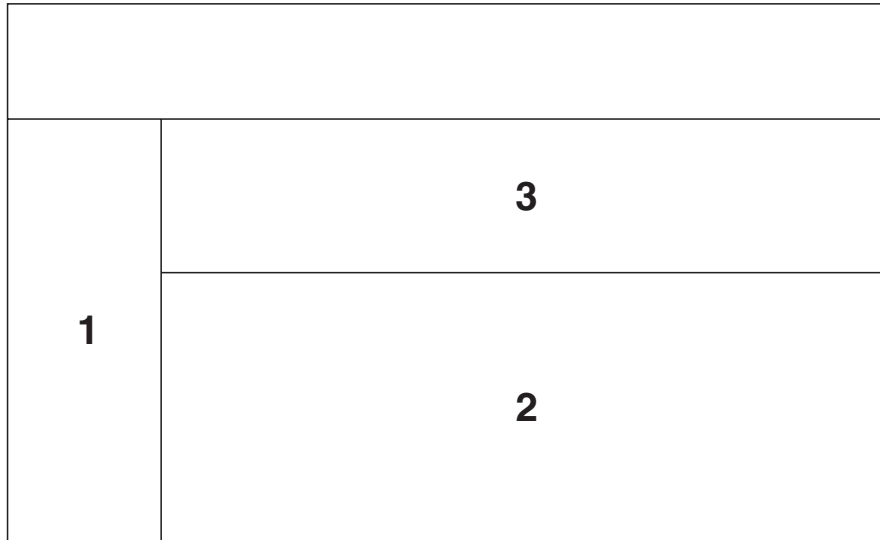
1 - Resource Selection Panel
2 - Counter Selection panel
3 - Viewing Counter (chart and table views)

Layout of the console

The performance viewer main window consists of two panels: the Resource Selection panel and the Data Monitoring panel. The Resource Selection panel, located on the left, provides a view of resources for which performance data can be displayed. The Data Monitoring panel, located on the right, displays numeric and statistical data for the resources that are highlighted (selected) in the Resource Selection panel.

You can adjust the width of the Resource Selection and Data Monitoring panels by dragging the split bar left or right. You can rearrange the order of the table columns in the Data Monitoring panel by dragging the column heading left or right. You can also adjust the width of the columns by dragging the edge of the column left or right.

Resource selection panel

The Resource Selection panel provides a hierarchical (tree) view of resources and the types of performance data available for those resources. Use this panel to select which resources to monitor and to start and stop data retrieval for those resources.

The Resource Selection panel displays resources and associated resource categories in an indented tree outline. Clicking the plus (+) and minus (-) symbols expands and collapses the tree to reveal the categories for the various resource instances. The resource tree can also be navigated by using the up and down arrow keys to cycle through the branches and by using the left and right arrow keys to expand and collapse the tree of resources. Resource instances can be expanded to reveal the instances they contain, if applicable. For example, when a EJB JAR instance is expanded, the enterprise bean instances in the EJB JAR are revealed. The Data Monitoring panel automatically displays the appropriate selection of counters for any objects highlighted in the Resource Selection panel.

The first level of the hierarchy includes all nodes (machines) in the administrative domain, followed by all application servers on the node. Below each application server, all resource categories are listed. If the enterprise beans category is expanded, all EJB JAR instances in the server are displayed. Next, all enterprise bean instances appear below the EJB JAR in the hierarchy. Then, a methods resource is associated with each bean. Clicking an individual bean or EJB JAR instance causes its corresponding counters to be displayed in the Data Monitoring panel. For enterprise beans, the counters displayed depend on whether the bean is an entity bean or a session bean. For EJB JARs, the counters are aggregate counters for all enterprise beans in the EJB JARs. See the InfoCenter article Performance data organization for more information.

Data monitoring panel

The Data Monitoring panel enables the selection of multiple counters and displays the resulting performance data for the currently selected resource. It contains two panels: the Viewing Counter panel above and the Counter Selection panel below.

Counter selection panel

The Counter Selection panel shows the counters available for the resource performance category selection.

Two factors determine the list of available counters in the Counter Selection panel:
- Only counters associated with the resource that is selected in the Resource Selection panel are displayed.
- Only counters having impact cost ratings within or below the instrumentation or monitoring level that is set for that resource in the administrative domain are displayed.

The first three counters shown for each resource performance category are selected by default. All counters can be selected or deselected, and the resulting output, shown in the top panel, automatically reflects the selection.

The columns in the Counter Selection panel supply the following information for each counter:
- **Name**. The names of the counters that are available for selection with this resource.
- **Description**. A brief description of the function of each counter.
- **Value**. The value for the counter, displayed according to the display mode in effect. Values are actual values (not scaled values used for the chart, if applicable).
- **Select**. A check box that indicates whether a counter is to be reflected in the chart. To hide data, clear the check box. The column representing that counter is then removed from the View Data window, and the graphic display for that counter is removed from the View Chart window.
- **Scale**. A value indicating whether data has been scaled (amplified or diminished) from its actual value to fit on the chart. This value is reflected only in the View Chart window.

  The value for the **Scale** column can be set manually by editing the value of the **Scale** field. See Scaling the chart display manually for information on manually setting the scale.

Viewing Counter panel

When a counter on the list in the Counter Selection panel is selected, the statistics gathered from that counter are displayed in the Viewing Counter panel at the top of the Data Monitoring panel.

The View Data window shows the counter's output in table format; the View Chart window displays a graph with time represented on the *x-axis* and the performance value represented on the *y-axis*. One or more performance counters can be simultaneously graphed on a single chart. The chart plots data from *n* data points, where *n* is the current table size (number of rows).

Display of multiple resources and aggregate data

When a single resource is selected in the Resource Selection panel, the Data Monitoring panel displays a choice of a table view or a chart view. If multiple resources are selected, the Data Monitoring panel displays a single data sheet for viewing summary information for the selected resources. The data sheet displays the tables for all objects of similar type for the selected resources. For example, if three servlet instances are selected, the data sheet displays a table of counter values for all the servlets. By default, the display buffer size is set to 40 rows, corresponding to the values of the last 40 data points retrieved.

The performance viewer provides aggregate data at the module level. If aggregate data is available for a group, it is displayed in the Data Monitoring panel. For example, for each enterprise bean home interface, counters track the number of active enterprise beans of that home. Each EJB JAR has an aggregate value that is the sum of all the enterprise beans in that EJB JAR. The enterprise beans resource category (module) within the application server has an aggregate value that is the sum of all enterprise beans in all EJB JARs.

Menu bar

The menu bar contains the following options:
- **File** menu. Used to change to current mode (from logging mode), to open an existing log file, and to exit from the performance viewer. The **File** menu contains the following items:
  - **Refresh**. Queries the administrative server for any newly started resources since data retrieval began or for additional counters to report. This operation is also recursive over all components subordinate to the selected resources. Tivoli Performance Viewer refreshes data every 10 seconds. When changing the refresh rate, you must use an integer greater than or equal to 1.
  - **Current Activity**. Resumes the display of real-time data in tables and charts. This menu option is used to stop viewing data from a log file and return to viewing real-time data.
  - **Log**. Displays a dialog box for specifying the name and location of an existing log file to be replayed.
  - **Exit**. Closes the performance viewer. If you made changes to the instrumentation levels of any resources during the session, a dialog box opens to ask whether you want to save the changed settings before closing the tool.
- **Logging** menu. Provides **On** and **Off** options that are used to start and stop recording data in a log file. If you start a new log file and specify the same file name, the file is overwritten.
- **Setting** menu. Used to start and stop the reporting of data, and to clear and refresh data. The **Setting** menu contains the following items:

- **Clear Buffer**. Deletes the values currently displayed in tables and charts. For example, after stopping a counter, you can use this operation to remove the remaining data from a table.
- **Reset to Zero**. Resets cumulative counters of the selected performance group back to zero.
- **View Data As**. Specifies how counter values are displayed. You can choose whether to display absolute values, changes in values, or rates of change. How data is displayed differs slightly depending on where you are viewing data. The choices follow:
    - **Raw Value**. Displays the absolute value. If the counter represents load data, such as the average number of connections in a database pool, then the Tivoli Performance Viewer displays the current value followed by the average. For example, 18 (avg:5).
    - **Change in Value**. Displays the change in the current value from the previous value.
    - **Rate of Change**. Displays the ratio $change/(T1 - T2)$, where $change$ is the change in the current value from the previous value, $T1$ is the time when the current value was retrieved and $T2$ is the time when the previous value was retrieved.
- **Log Replay**. Includes **Rewind Stop Play Fast Forward** .

Note that right-clicking a resource in the Resource Selection panel displays a menu that provides the following options: **Refresh**, **Clear Buffer**, and **Reset to Zero**.

- **Help** menu. Provides information for users.

Toolbar icons

Toolbar icons provide shortcuts to frequently used commands. The toolbar includes the following icons:

- **Refresh**. Updates data and structures for the selected resources. That is, it polls the administrative server to retrieve new information about additional counters to display or new servers recently added to the domain.
- **Clear Buffer**. Deletes the values currently displayed in all tables and charts.
- **Reset to Zero**. Resets the counters.

Node icons

In the Resource Selection panel, the color of the node icon indicates the current state and availability of the application server in the domain.

- Green—The resource is running and available.
- Red—The resource is stopped.

Status bar

The status bar across the bottom of the performance viewer window dynamically displays the current state of the reporting values. The following state information is reported in the status bar:

- The current setting for the refresh rate
- The buffer size in use in the current Viewing Counter panel
- The display mode in use in the current Viewing Counter panel
- The current state of the logging setting

Starting the Tivoli Performance Viewer

An alternative way to collect data is to use the Tivoli Performance Viewer, a Graphical User Interface (GUI) performance monitor shipped with WebSphere Application Server. You can also start the Tivoli Performance Viewer with security enabled. To do this see Running your monitoring applications with security enabled.

Steps for this task

1. Enable PMI services through the administrative console.
2. Start the Tivoli Performance Viewer.

   This can be done in two ways:

   a. Start performance monitoring from the command line.

      Go to the `<i>product_installation_directory</i>`/bin directory and run the `tperfviewer` script.

      You can specify the host and port in Windows NT and 2000 environments as:

      `tperfviewer.bat` *host_name port_number connector_type*

      or

      On the AIX and other UNIX platforms, use

      `tperfviewer.sh` *host_name port_number connector_type*

      for example:

      `tperfviewer.bat localhost 8879 SOAP`
      Connector_type can be either SOAP or RMI.
      8879 is the default ND port for SOAP connector.
      9809 is the default ND port for RMI connector

   b.

      Click **Start** > **Programs** > **IBM WebSphere** > **Application Server v.50** > **Tivoli Performance Viewer**.

      Tivoli Performance Viewer detects which package of WebSphere Application Server you are using and connects using the default Remote Method Invocation (RMI) connector port. If the connection fails, a dialog is displayed to provide new connection parameters.

      You can connect to a remote host or a different port number, by using the command line to start the performance viewer.

3. Adjust the data collection settings.

   Refer to the instructions in the topic "Setting performance monitoring levels".

Setting performance monitoring levels

Before you begin

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

To view monitoring settings:

Steps for this task

1. Choose the **Data Collection** icon on the Resource Selection panel.

This selection provides two options on the Counter Selection panel. Choose the **Current Activity** option to view and change monitoring settings. Alternatively, use **File**> **Current Activity** to view the monitoring settings.

2. Set monitoring levels by choosing one of the following options:
   - **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

   These options apply to an entire application server.

3. (Optional) Fine tune the monitoring level settings.
   a. Click **Specify**.

      This sets the monitoring level to custom.
   b. Select a monitoring level.

      For each resource, choose a monitoring level of **None**, **Low**, **Medium**, **High** or **Maximum**. The dial icon will change to represent this level. **Note:** The instrumentation level is set recursively to all elements below the selected resource. You can override this by setting the levels for children AFTER setting their parents.

4. Click **OK**.
5. Click **Apply**.

Results

If the instrumentation level excludes a counter, that counter does not appear in the tables and charts of the performance viewer. For example, when the instrumentation level is set to low, the thread pool size is not displayed because that counter requires a level of high.

Note that monitoring levels can also be set through the administrative console. See the InfoCenter article Enabling data collection through the administrative console for more information.

Setting monitoring levels for individual enterprise bean methods
Before you begin

Due to performance overhead, the **Standard** monitoring level does not include monitoring individual enterprise bean methods.

To monitor individual methods:

Steps for this task
1. Choose the **Custom** option for setting monitoring levels.
2. Set the monitoring level for the methods category to **Maximum** by following the procedure described in setting the monitoring level task.
3. Click **Apply**.
4. Click **OK**.

Results

Individual methods display, and you can set the level for individual methods.

Only methods called by an application display. If a remote method has not been called since the application server started, it does not appear in the performance panel.

Viewing summary reports

<u>Before you begin</u>

Summary reports are available for each application server. Before viewing reports, make sure data counters are enabled and monitoring levels are set properly.

The standard monitoring level will enable 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:

<u>Steps for this task</u>
1. Click the application server icon in the navigation tree.
2. Click the appropriate column header to sort the columns in the report.

Changing the refresh rate of data retrieval

<u>Before you begin</u>

By default, the Tivoli Performance Viewer retrieves data every 10 seconds.

To change the rate at which data is retrieved:

<u>Steps for this task</u>
1. Click **Setting** > **Set Refresh Rate**.
2. Type a positive integer representing the number of seconds in the **Set Refresh Rate** dialog box.
3. Click **OK**.

Changing the display buffer size

To change the size of the buffer and the number of rows displayed:

<u>Steps for this task</u>
1. Click **Setting** > **Set Buffer Size**.
2. Type the number of rows to display in the **Set Buffer Size** dialog box.
3. Click **OK**.

Viewing and modifying performance chart data

<u>Before you begin</u>

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 a chart:

<u>Steps for this task</u>
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 set the scaling factors by typing directly in the scale field. See Scaling the performance data chart display for more information.

Scaling the performance data chart display

You can manually adjust the scale for counters so that the graphic displays enable meaningful comparisons between graphs of different counters. Follow these steps to manually adjust the scale:

Steps for this task

1. Double-click the **Scale** column for the counter that you want to modify.
2. Type the desired value in the field for the **Scale** value.

The **View Chart** display immediately reflects the change in the scaling factor.

Results

The possible values for the **Scale** field range from 0 to 100 and show the following relationships:

- A value equal to 1 indicates that the value is the actual value. The value represents the default setting.
- A value greater than 1 indicates that the variable value is amplified by the factor shown. For example, a scale setting of 1.5 means that the the variable is graphed as one and one-half times their actual values.
- A value less than 1 indicates that the variable value is decreased by the factor shown. For example, a scale setting of .5 means that the the variable is graphed as one-half its actual values.

Scaling only applies to the graphed values.

Refreshing data

The refresh operation is a local, not global, operation that applies only to selected resources. The refresh operation is recursive; all subordinate or children resources refresh when a selected resource refreshes. To refresh data:

Steps for this task

1. Click one or more resources in the Resource Selection panel.
2. Click **File** > **Refresh**. Alternatively, click the **Refresh** icon or right-click the resource and select **Refresh**.

Clicking refresh with server selected under the viewer icon causes TPV to query the server for new PMI and product configuration information. Clicking refresh with server selected under the advisor icon causes TPV to refresh the advice provided, but will not refresh PMI or product configuration information.

Performance data refresh behavior

New performance data can become available in either of the following situations:

- An administrator uses the console to change the instrumentation level for a resource (for example, from medium to high).
- An administrator uses the console to add a new resource (for example, an enterprise bean or a servlet) to the run time.

In both cases, if the resource in question is already polled by the Tivoli Performance Viewer or the parent of the resource is being polled, the system is automatically refreshed. If more counters are added for a group that the performance viewer is already polling, the performance viewer automatically adds the counters to the table or chart views. If the parent of the newly added resource is polled, the new resource is detected automatically and added to the Resource Selection tree. You can refresh the Resource Selection tree, or parts of it, by selecting the appropriate node and clicking the **Refresh** icon, or by right-clicking a resource and choosing **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.

Clicking refresh with server selected under the viewer icon causes TPV to query the server for new PMI and product configuration information. Clicking refresh with server selected under the advisor icon causes TPV to refresh the advice provided, but will not refresh PMI or product configuration information.

Clearing values from tables and charts

Before you begin

Selecting **Clear Values** removes remaining data from a table or chart. You can then begin populating the table or chart with new data.

To clear the values currently displayed:

Steps for this task
1. Click one or more resources in the Resource Selection panel.
2. Click **Setting** > **Clear Values**. Alternatively, right-click the resource and select **Clear Values**

Storing data to a log file

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

To start recording data:

Steps for this task
1. Click **Logging** > **On** or click the **Logging** icon.
2. Specify the name, location, and format type of the log file in the **Save** dialog box.

   The **Files of type** field allows an extension of `*.perf` for binary files or `*.xml` for XML format. The XML format provides a flexibility that enables analysis by using third-party tools.

   Note: The `*.perf` files may not be compatible between fix levels.
3. Click **OK**.

What to do next

To stop logging, click **Logging** > **Off** or click the **Logging** icon.

Performance data log file

An example of the performance data log file format is below.

Location

By default, this file is written to:

product_installation_root/logs/ra_mmdd_hhmm.xml

where mmdd=month and date, and hhmm=hour and minute

Usage Notes

This read-write data file is created by Tivoli Performance Viewer and provides data collected by the performance viewer. The log file is not updated, but remains available for you to replay the collected data. The performance data log file does not have an effect on the WebSphere environment.

Example

```
<?xml version="1.0"?>
<RALog version="5.0">
<RAGroupSnapshot time="1019743202343" numberGroups="1">
  <CpdCollection name="root/peace/Default Server/jvmRuntimeModule" level="7">
   <CpdData name="root/peace/Default
            Server/jvmRuntimeModule/jvmRuntimeModule.total/Memory" id="1">
    <CpdLong value="39385600" time="1.019743203334E12"/>
   </CpdData>
   <CpdData name="root/peace/Default
         Server/jvmRuntimeModule/jvmRuntimeModule.freeMemory" id="2">
    <CpdLong value="4815656" time="1.019743203334E12"/>
   </CpdData>
   <CpdData name="root/peace/Default
         Server/jvmRuntimeModule/jvmRuntimeModule.usedMemory" id="3">
    <CpdLong value="34569944" time="1.019743203334E12"/>
   </CpdData>
  </CpdCollection>
 </RAGroupSnapshot>
</RALog>
```

Replaying a performance data log file

Before you begin

You can replay both binary and XML logs by using the Tivoli Performance Viewer.

To replay a log file, do the following:

Steps for this task

1. Click **Data Collection** in the navigation tree.
2. Click the **Log** radio button in the **Performance data from** field.
3. Click **Browse** to locate the file that you want to replay or type the file pathname in the **Log** field.
4. Click **Apply**.
5. Play the log by using the **Play** icon or click **Setting** > **Log Replay** > **Play**.

Results

By default, the data replays at the same rate it was collected or written to the log. If data is collected every minute, it displays every minute. You can choose **Fast**

Forward mode in which the log replays without simulating the refresh interval. To **Fast Forward**, use the button in the tool bar or click **Setting** > **Log Replay** > **Fast Forward**.

To rewind a log file, click **Setting** > **Log Replay** > **Rewind** or use the **Rewind** icon in the toolbar.

While replaying the log, you can choose different groups to view by selecting them in the Resource Selection pane. You can also view the data in either of the views available in the tabbed Data Monitoring panel.

You can stop and resume the log at any point. However, you cannot replay data in reverse.

Resetting counters to zero

Some counters report relative values based on how much the value has changed since the counter was enabled. The **Reset to Zero** operation resets those counters so that they will report changes in values since the reset operation. This operation will also clear the buffer for the selected resources. See "Clearing values from tables and charts" in Related Links for more information about clearing the buffer for selected resources. Counters based on absolute values can not be reset and will not be affected by the **Reset to Zero** operation.

To reset the start time for calculating relative counters:

Steps for this task

1. Click one or more resources in the Resource Selection panel.
2. Click **Setting** > **Reset to Zero**. Alternatively, right-click the resource and click **Reset to Zero**.

Tivoli performance monitoring and management solutions

Tivoli offers the complete IBM solution for managing the extended WebSphere environment. For precise viewing of performance metrics, users can start with the Tivoli Performance Viewer, a complimentary tool shipped with WebSphere Application Server.

Tivoli also provides on-going production monitoring tools described below. For more information about Tivoli's solutions for WebSphere Application Server, see the InfoCenter article Performance: Resources for Learning.

IBM Tivoli Monitoring for Web Infrastructure (ITMf WI). Provides best-practice monitoring of the key elements of WebSphere Application Server. This is the inside-out view, enabling administrators to quickly address problems before they impact end-users. Using Tivoli's advanced monitoring technology and predefined WebSphere best-practices, this tool quickly identifies problems, notifies appropriate personnel, and provides a solution. All monitoring data is displayed real-time with a health console displaying non-stop data. This same information can be uploaded to a common data warehouse for historical reporting.

IBM Tivoli Monitoring for Transaction Performance (ITMTP). Provides a unique monitoring perspective from that of the end-user. This is the outside-in view that verifies that end-to-end components provide a positive end-user experience. ITMTP monitors performance of actual and synthetic transactions, as well as verifying that the content delivered meets predefined guidelines.

Transaction performance includes total round trip response time, network latency, back-end response time and page render time. Additional granularity of transaction detail on the back-end is provided through Application Response Measurement instrumentation.

The ITM and ITMTP function by providing Web site performance monitoring, alerting customers to end user response time issues.

The ability to quickly find performance issues is key to maintaining a high performance Web site. This WebSphere Application Server release and the new ITMTP release combine to provide a new feature for analyzing performance problems. Using Synthetic Transaction Investigator (STI) from ITMTP, you can save key transactions and replay them later. ITMTP also collects the data provided by PMI Request Metrics through the Application Response Measurement (ARM) interface and correlates this information with the originating STI transaction. In the ITMTP real-time browser, the STI information links to the servlet and the enterprise bean response time data. The details regarding the overall transaction response time and response time for individual WebSphere Application Server components provide the ability to quickly identify performance problems.

Tivoli provides additional products for monitoring other key elements of the extended environment. For more information about Tivoli's solutions for WebSphere Application Server, see the topic "Performance: Resources for Learning".

Developing your own monitoring applications

Before you begin

You can use the Performance Monitoring Infrastructure (PMI) interfaces to develop your own applications to collect and display performance information.

There are three such interfaces - a Java Machine Extension (JMX)-based interface, a PMI client interface, and a servlet interface. All three interfaces return the same underlying data. The JMX interface is accessible through the AdminClient tool. The PMI client interface is a Java interface that works with Version 3.5.5 and above. The servlet interface is perhaps the simplest, requiring minimal programming, as the output is XML.

Steps for this task
1. Developing your own monitoring application using Performance Monitoring Infrastructure client .
2. Developing your own monitoring applications with PMI servlet
3. Compiling your monitoring applications
4. Running your new monitoring applications
5. Accessing Performance Monitoring Infrastructure data through the Java Management Extension interface.
6. Developing Performance Monitoring Infrastructure interfaces (Version 4.0).

Developing your own monitoring application using Performance Monitoring Infrastructure client

The following is the programming model for Performance Monitoring Infrastructure (PMI) client:

<u>Steps for this task</u>

1. Create an instance of PmiClient.

   This is used for all subsequent method calls.

2. Call the listNodes() and listServers(nodeName) methods to find all the nodes and servers in the WebSphere Application Server domain.

   The PMI client provides two sets of methods: one set in Version 5.0 and the other set inherited from Version 4.0. You can only use one set of methods. Do not mix them together.

3. Call listMBeans and listStatMembers to get all 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.

<u>What to do next</u>

If you prefer to use the Version 4.0 version of the interface, the model is essentially the same, but the object types are different:

1. Create an instance of PmiClient.

2. Call the listNodes() and listServers(nodeName) methods to find all the nodes and servers in the WebSphere Application Server domain.

3. Call listMembers to get all the perfDescriptor objects.

4. Use the PMI client's get or gets method to get CpdCollection objects. These contain snapshots of 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.

Performance Monitoring Infrastructure 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 WebSphere Application Server 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.

Performance Monitoring Infrastructure client interface

The data provided by the Performance Monitoring Infrastructure (PMI) client interface is documented here. Access to the data is provided in a hierarchical structure. Descending from the object are node information objects, module information objects, CpdCollection objects and CpdData objects. Using Version 5.0, you will get Stats and Statistic objects. The node and server information objects

contain no performance data, only static information.

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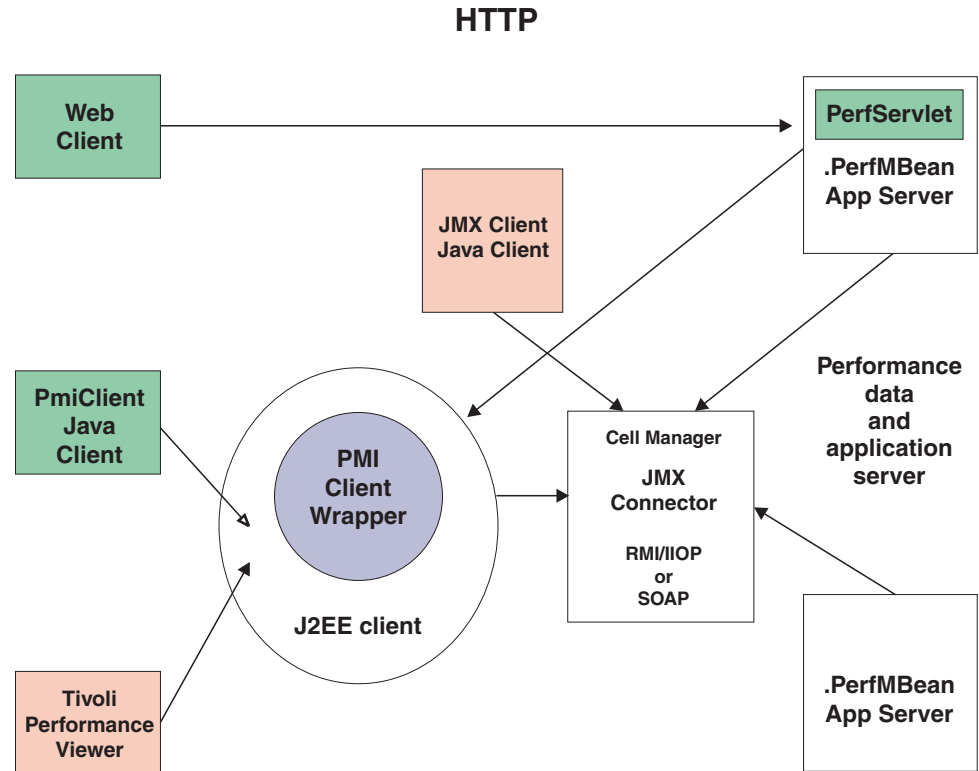
Each time a client retrieves performance data from a server, the data is returned in a subset of this structure; the form of the subset depends on the data retrieved. You can update the entire structure with new data, or update only part of the tree, as needed.

The JMX statistic data model is supported, as well as the existing CPD data model from Version 4.0. When you retrieve performance data using the Version 5.0 PMI client API, you get the Stats object, which includes Statistic objects and optional sub-Stats objects. When you use the Version 4.0 PMI client API to collect performance data, you get the CpdCollection object, which includes the CpdData objects and optional sub-CpdCollection objects.

The following are additional Performance Monitoring Infrastructure (PMI) interfaces:
- BoundaryStatistic
- BoundedRangeStatistic
- CountStatistic
- MBeanStatDescriptor
- MBeanLevelSpec
- New Methods in PmiClient
- RangeStatistic
- Stats
- Statistic
- TimeStatistic

The following PMI interfaces introduced in Version 4.0 are also supported:

- CpdCollection
- CpdData
- CpdEventListener and CpdEvent
- CpdFamily class
- CpdValue
  - CpdLong
  - CpdStat
  - CpdLoad
- PerfDescriptor
- PmiClient class

The CpdLong maps to CountStatistic; CpdStat maps to Time Statistic; CpdCollection maps to Stats; and CpdLoad maps to RangeStatistic and BoundedRangeStatistic.

Note: Version 4.0 PmiClient APIs are supported in this version, however, there are some changes. The data hierarchy is changed in some PMI modules, notably the enterprise bean module and HTTP sessions module. If you have an existing PmiClient application, and you want to run it against Version 5.0, you might have to update the PerfDescriptor(s) based on the new PMI data hierarchy. Also, the getDataName and getDataId methods in PmiClient are changed to be non-static methods in order to support multiple WebSphere Application Server versions. You might have to update your existing application which uses these two methods.

Using Version 5.0 PMI API in Version 3.5.5+ and Version 4.0.x

For Version 3.5.5+, follow these instructions:

Steps for this task
1. Make configuration changes.

   For PMI to interact correctly with Version 3.5.x application servers, you must upgrade both the Version 3.5.x run time environment and the PMI JAR files to the levels specified below. In addition, you must prepend the `repository.jar`, `ejs.jar`, and `ujc.jar` files from the upgraded Version 3.5.x run time environment to the PMI client's run time classpath.

   a. Change the Version 3.5.x run time environment.

      Ensure the Version 3.5 environment is Version 3.5.5 or later.

   b. Change the PMI client's run time or development environment.

      Both the Version 5.0 PMI client and the Version 4.02 client can work with the Version 3.5.5+ WebSphere Application Server.

      Copy the `repository.jar`, `ujc.jar` and `ejs.jar` files from the *WebSphere_35_installation_root*`/lib directory` to each machine from which a PMI client is run.

      Prepend the Version 3.5.5+ `repository.jar`, `ujc.jar` and `ejs.jar` files to the PMI client's run time classpath.

2. Copy the XML configuration files from Version 4.0.2+.

   a. Get the `perf.jar` file from Version 4.0.

   b. Append the `perf.jar` file to the classpath of the Version 5.0 PMI client.

      Note: Ensure the Version 5.0 `pmi.jar` file and `pmiclient.jar` files come **before** the Version 4.0 `perf.jar` file.

3. Make programmatic changes.

A new constructor for PmiClient allows a client to monitor Version 3.5.5 or later application servers. The new constructor takes three string parameters: hostName, serverName, and version.

```
public PmiClient(String host, String port, String version)
```

Using this constructor with "EPM" as the third parameter creates a PmiClient that can retrieve data from Version 3.5.5+ application servers.

```
PmiClient pmiClnt = new PmiClient(hostName, portNumber, "EPM")
```

What to do next

Use Version 4.0 PmiClient API to write your own client application with WebSphere Application Server Version 4.0 and 3.5.5+. See the example code for using Version 4.0 API in the topic "Example: Performance Monitoring Infrastructure client (Version 4.0)".

To run a Version 5.0 PMI client with a Version 4.0 server, the instructions are similar, except in substep 2 of step 1, you need to copy the repository.jar and ujc.jar files from a WebSphere Application Server, Version 4.0, installation.

Example: Performance Monitoring Infrastructure client (Version 4.0)

The following is a list of example Performance Monitoring Infrastructure (PMI) client code from Version 4.0:

```
/**
 * This is a sample code to show how to use PmiClient to collect PMI data.
 * You will need to use adminconsole to set instrumentation level (a level other
 * than NONE) first.
 *
 * <p>
 * End-to-end code path in 4.0:
 *     PmiTester -> PmiClient -> AdminServer -> appServer
 */


package com.ibm.websphere.pmi;

import com.ibm.websphere.pmi.*;
import com.ibm.websphere.pmi.server.*;
import com.ibm.websphere.pmi.client.*;
import com.ibm.ws.pmi.server.*;
import com.ibm.ws.pmi.perfServer.*;
import com.ibm.ws.pmi.server.modules.*;
import com.ibm.ws.pmi.wire.*;
import java.util.ArrayList;


/**
 * Sample code to use PmiClient API (old API in 4.0) and get CpdData/CpdCollection objects.
 *
 */
public class PmiTester implements PmiConstants {

    /** a test driver:
     * @param args[0] - node name
     * @param args[1] - port number, optional, default is 2809
     * @param args[2] - connector type, default is RMI
     * @param args[3] - verion (AE, AEs, WAS50), default is WAS50
     *
     */
    public static void main(String[] args) {
        String hostName = null;
```

```
            String portNumber = "2809";
            String connectorType = "RMI";
            String version = "WAS50";

            if (args.length < 1) {
                System.out.println("Usage: <host> [<port>] [<connectorType>]
[<version>]");
                return;
            }

            if(args.length >= 1)
                hostName = args[0];
            if(args.length >= 2)
                portNumber = args[1];
            if (args.length >=3)
                connectorType = args[2];
            if (args.length >=4)
                version = args[3];

            try {
                PmiClient pmiClnt = new PmiClient(hostName, portNumber,
version, false, connectorType);

                // uncomment it if you want debug info
                //pmiClnt.setDebug(true);

                // get all the node PerfDescriptor in the domain
                PerfDescriptor[] nodePds = pmiClnt.listNodes();

                if(nodePds == null) {
                    System.out.println("no nodes");
                    return;
                }

                // get the first node
                String nodeName = nodePds[0].getName();
                System.out.println("after listNodes: " + nodeName);

                //list all the servers on the node
                PerfDescriptor[] serverPds = pmiClnt.listServers(nodePds[0].getName());
                if(serverPds == null || serverPds.length == 0) {
                    System.out.println("NO app server in node");
                    return;
                }

                // print out all the servers on that node
                for(int j=0; j<serverPds.length; j++) {
                    System.out.println("server " + j + ": " + serverPds[j].getName());
                }

                for(int j=0; j<serverPds.length; j++) {
                    System.out.println("server " + j + ": " + serverPds[j].getName());

                    // Option: you can call createPerfLevelSpec and then
setInstrumentationLevel to set the level
                    // for each server if you want. For example, to set all
the modules to be LEVEL_HIGH for the server j,
                    // uncomment the following.
                    // PerfLevelSpec[] plds = new PerfLevelSpec[1];
                    // plds[0] = pmiClnt.createPerfLevelSpec(null, LEVEL_HIGH);
                    // pmiClnt.setInstrumentationLevel(serverPds[j].getNodeName(),
serverPds[j].getServerName(), plds, true);

                    // First, list the PerfDescriptor in the server
                    PerfDescriptor[] myPds = pmiClnt.listMembers(serverPds[j]);

                    // check returned PerfDescriptor
```

```
                    if(myPds == null) {
                        System.out.println("null from listMembers");
                        continue;
                    }

                    // you can  add the pds in which you are interested to PerfDescriptorList
                    PerfDescriptorList pdList = new PerfDescriptorList();
                    for(int i=0; i<myPds.length; i++) {
                        // Option 1: you can recursively call listMembers for each myPds
                        //            and find the one you are interested. You can call
listMembers
                        //            until individual data level and after that level
you will null from listMembers.
                        // e.g., PerfDescriptor[] nextPds = pmiClnt.listMembers(myPds[i]);

                        // Option 2: you can filter these pds  before adding to pdList
                        System.out.println("add to pdList: " + myPds[i].getModuleName());
                        pdList.addDescriptor(myPds[i]);
                        if( i % 2 == 0)
                            pmiClnt.add(myPds[i]);
                    }

                    // call gets method to get the CpdCollection[] corresponding to pdList
                    CpdCollection[] cpdCols = pmiClnt.gets(pdList, true);

                    if(cpdCols == null) {
                        // check error
                        if(pmiClnt.getErrorCode() >0)
                            System.out.println(pmiClnt.getErrorMessage());
                        continue;
                    }

                    for(int i=0; i<cpdCols.length; i++) {
                        // simple print them
                        //System.out.println(cpdCols[i].toString());

                        // Or call processCpdCollection to get each data
                        processCpdCollection(cpdCols[i], "");
                    }

                    // Or call gets() method to add the CpdCollection[] for whatever
there by calling pmiClnt.add().
                    System.out.println("\n\n\n ---- get data using gets(true) ----- ");
                    cpdCols = pmiClnt.gets(true);

                    if(cpdCols == null) {
                        // check error
                        if(pmiClnt.getErrorCode() >0)
                            System.out.println(pmiClnt.getErrorMessage());
                        continue;
                    }

                    for(int i=0; i<cpdCols.length; i++) {
                        // simple print out the whole collection
                        System.out.println(cpdCols[i].toString());

                        // Option: refer processCpdCollection to get each data
                    }
                }

            }
            catch(Exception ex) {
                System.out.println("Exception calling CollectorAE");
                ex.printStackTrace();
            }
        }
```

```
/**
 * show the methods to retrieve individual data
 */
private static void processCpdCollection(CpdCollection cpdCol, String indent) {
    CpdData[] dataList = cpdCol.dataMembers();
    String myindent = indent;

    System.out.println("\n" + myindent + "--- CpdCollection "
+ cpdCol.getDescriptor().getName() + " ---");
    myindent += "   ";
    for(int i=0; i<dataList.length; i++) {
        if (dataList[i] == null)
            continue;

        // if you want to get static info like name, description, etc
        PmiDataInfo dataInfo = dataList[i].getPmiDataInfo();
        // call getName(), getDescription() on dataInfo;

        CpdValue cpdVal = dataList[i].getValue();
        if(cpdVal.getType() == TYPE_STAT) {
            CpdStat cpdStat = (CpdStat)cpdVal;
            double mean = cpdStat.mean();
            double sumSquares = cpdStat.sumSquares();
            int count = cpdStat.count();
            double total = cpdStat.total();
            System.out.println(myindent + "CpdData id=" + dataList[i].getId()
                            + " type=stat mean=" + mean);
            // you can print more values like sumSquares, count,etc here
        }
        else if(cpdVal.getType() == TYPE_LOAD) {
            CpdLoad cpdLoad = (CpdLoad)cpdVal;
            long time = cpdLoad.getTime();
            double mean = cpdLoad.mean();
            double currentLevel = cpdLoad.getCurrentLevel();
            double integral = cpdLoad.getIntegral();
            double timeWeight = cpdLoad.getWeight();
            System.out.println(myindent + "CpdData id=" + dataList[i].getId()
                            + " type=load mean=" + mean + " currentLevel="
+ currentLevel);
            // you can print more values like sumSquares, count,etc here
        }
        else if(cpdVal.getType() == TYPE_LONG) {
            CpdValue cpdLong = (CpdValue)cpdVal;
            long value = (long)cpdLong.getValue();
            System.out.println(myindent + "CpdData id=" + dataList[i].getId()
                            + " type=long value=" + value);
        }
        else if(cpdVal.getType() == TYPE_DOUBLE) {
            CpdValue cpdDouble = (CpdValue)cpdVal;
            double value = cpdDouble.getValue();
            System.out.println(myindent + "CpdData id=" + dataList[i].getId()
                            + " type=double value=" + value);
        }
        else if(cpdVal.getType() == TYPE_INT) {
            CpdValue cpdInt = (CpdValue)cpdVal;
            int value = (int)cpdInt.getValue();
            System.out.println(myindent + "CpdData id=" + dataList[i].getId()
                            + " type=int value=" + value);
        }
    }

    // recursively go through the subcollection
    CpdCollection[] subCols = cpdCol.subcollections();
    for(int i=0; i<subCols.length; i++) {
        processCpdCollection(subCols[i], myindent);
    }
}
```

```
    /**
     * show the methods to navigate CpdCollection
     */
    private static void report(CpdCollection col) {
        System.out.println("\n\n");
        if(col==null) {
            System.out.println("report: null CpdCollection");
            return;
        }
        System.out.println("report - CpdCollection ");
        printPD(col.getDescriptor());
        CpdData[] dataMembers = col.dataMembers();
        if(dataMembers != null) {
            System.out.println("report CpdCollection: dataMembers is "
+ dataMembers.length);
            for(int i=0; i<dataMembers.length; i++) {
                CpdData data = dataMembers[i];
                printPD(data.getDescriptor());
            }
        }
        CpdCollection[] subCollections = col.subcollections();
        if(subCollections != null) {
            for(int i=0; i<subCollections.length; i++) {
                report(subCollections[i]);
            }
        }
    }

    private static void printPD(PerfDescriptor pd) {
        System.out.println(pd.getFullName());
    }

}
```

Example: Performance Monitoring Infrastructure client with new data structure

The following is example code using Performance Monitoring Infrastructure (PMI) client with the new data structure:

```
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;
        }
    }


}
```

Developing your own monitoring applications with Performance Monitoring Infrastructure servlet

Before you begin

The performance servlet uses the Performance Monitor Interface (PMI) infrastructure to retrieve the performance information from WebSphere Application Server. This is the same infrastructure used by the Tivoli Performance Viewer and is subject to the same restrictions on the availability of data as the performance viewer.

The performance servlet `.ear` file `perfServletApp.ear` is located in the *install_root* directory.

The performance servlet is deployed exactly as any other servlet. To use it, follow these steps:

Steps for this task
1. Deploy the servlet on a single application server instance within the domain.
2. After the servlet deploys, you can invoke it to retrieve performance data for the entire domain.

   Invoke the performance servlet by accessing the following default URL:

   `http://hostname/wasPerfTool/servlet/perfservlet`

Results

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 the topic "Performance Monitoring Infrastructure (PMI) servlet" for more information about the PMI servlet output.

Performance Monitoring Infrastructure 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
* PerfStatInfo
* PerfLoadInfo

PerfNumericInfo. When each invocation of the performance servlet retrieves the performance values from Performance Monitoring Infrastructure (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 since 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 the following 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 the following 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 the following 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 since 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>
```

When the performance servlet is first initialized, it retrieves the list of nodes and servers located within the domain in which it is deployed. Because the collection of this data is expensive, the performance servlet holds this information as a cached list. If a new node is added to the domain or a new server is started, 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 as follows:

```
http://hostname/wasPerfTool/servlet/perfservlet?refreshConfig=true
```

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

- **Node.** The servlet can limit the information it provides to a specific host by using the node parameter. For example, to limit the data collection to the node rjones, invoke the following URL:

  ```
  http://hostname/wasPerfTool/servlet/perfservlet?Node=rjones
  ```

- **Server.** The servlet can limit the information it provides to a specific server by using the server parameter. For example, in order to limit the data collection to the TradeApp server on all nodes, invoke the following URL:

  ```
  http://hostname/wasPerfTool/servlet/perfservlet?Server=TradeApp
  ```
  To limit the data collection to the TradeApp server located on the host rjones, invoke the following URL:

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

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

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

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

  ```
  http://hostname/wasPerfTool/servlet/perfservlet?Module=beanModule
  ```
  To limit the data collection to the beanModule on the server TradeApp on the node rjones, invoke the following URL:

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

Compiling your monitoring applications

Before you begin

To compile your Performance Monitoring Infrastructure (PMI) code, you must have the following JAR files in your 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.

Running your new monitoring applications

Steps for this task

1. Obtain the `pmi.jar` and `pmiclient.jar` files.

   The `pmi.jar` and `pmiclient.jar` files are required for client applications using 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 it 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. Use PMI client API to write your own application.

3. Compile the newly written PMI application and place it on the classpath.

4. Run the application with the following script:

   ```
   call "%~dp0setupCmdLine.bat"

   set WAS_CP=%WAS_HOME%\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 name
   [port] [connectorType]
   ```

Performance Monitoring Infrastructure client package

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

As shown in the following figure, PmiClient uses the AdminClient API to communicate the Perf MBean in an application server.

PmiClient communicates with the network manager first, retrieving an AdminClient instance to each application server. Once the PmiClient receives the instance, it uses it to communicate with the application server directly for performance or level setting changes. Since level settings are persistent through PmiClient, you are only required to set it once, unless you want to change it.

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 using the PmiClient API, you have to pass the JMX connector protocol and port number to instantiate an object of the PmiClient. Once 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.
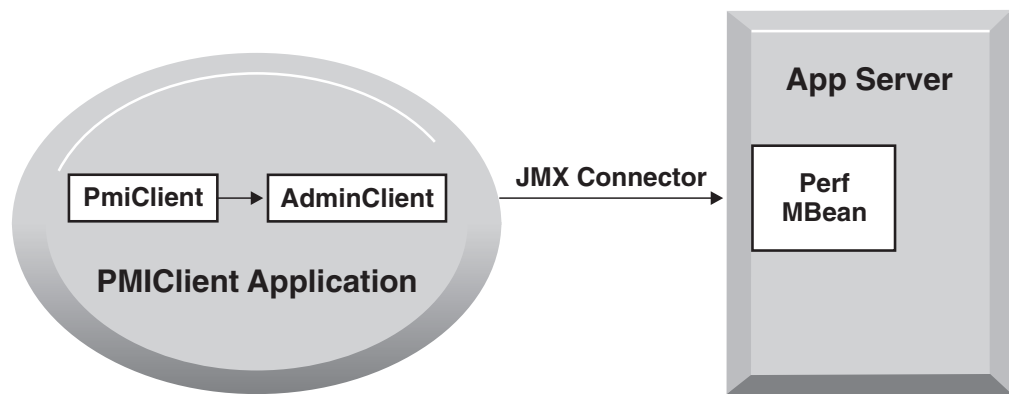
Running your monitoring applications with security enabled

In order to run a Performance Monitoring Infrastructure client application with security enabled, you must have %CLIENTSOAP% and %CLIENTSAS% properties on your Java virtual machine command line. The %CLIENTSOAP% and %CLIENTSAS% properties are defined in the setupCmdLine.bat or setupCmdline.sh files.

Steps for this task

1. Set com.ibm.SOAP.securityEnabled to **True** in the soap.client.props file for the SOAP connector.

   The soap.client.props property file is located in the WAS_ROOT/properties directory.

2. Set com.ibm.SOAP.loginUserid and com.ibm.SOAP.loginPassword as the user ID and password for login.

3. Set the sas.client.props file or type the user ID and password in the pop-up window if you do not put them in the property file for RMI connector

   A common mistake is leaving extra spaces at the end of the lines in the property file. Do not leave extra spaces at the end of the lines, especially for the user ID and password lines.

Accessing Performance Monitoring Infrastructure data through the Java Management Extension interface

Before you begin

WebSphere Application Server allows you to invoke methods on MBeans through the AdminClient Java Management Extension (JMX) interface. You can use

AdminClient API to get Performance Monitoring Infrastructure (PMI) data by using either PerfMBean or individual MBeans. See information about using individual MBeans at bottom of this article.

Individual MBeans provide the Stats attribute from which you can get PMI data. The 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. PMI can be delivered in a single JMX cell through PerfMBean, but multiple JMX calls have to be made through individual MBeans.

See the topic "Developing an administrative client program" for more information on AdminClient JMX.

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

Steps for this task

1. Create an instance of AdminClient.

   When using AdminClient API, you need to first create an instance of AdminClient by passing 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

   Once you get the AdminClient instance, you can call queryNames to get a list of MBean ObjectNames depending on your query string. To get all the ObjectNames, you can use the following example code. If you have a specified query string, you will 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 the following example code to get all 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 will only return nodes that are started.

To list running servers on the node, you can either check the node name and type for all the 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.

Once you get the PerfMBean(s) in the application server from which you want to get PMI data, you can invoke the following operations on the PerfMBean through AdminClient API:

```
- 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);
```

   Note: 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.

- 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);
```

   Note: 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.

- 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);
```

- **To use an individual MBean:** You need to get the AdminClient instance and
  the ObjectName for the individual MBean. Then you can simply get the Stats
  attribute on the MBean.

Example: Administering Java Management Extension-based interface
Examples

The following is example code directly using Java Management Extension (JMX)
API. For information on compiling your source code, see "Compiling your
monitoring applications."

```
package com.ibm.websphere.pmi;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.exception.ConnectorException;
import com.ibm.websphere.management.exception.InvalidAdminClientTypeException;
import com.ibm.websphere.management.exception.*;

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

/**
 * Sample code to use AdminClient API directly to get PMI data from PerfMBean
 * and individual MBeans which support getStats method.
 */

public class PmiJmxTest implements PmiConstants {
```

```
                private AdminClient   ac = null;
                private ObjectName    perfOName  = null;
                private ObjectName    serverOName = null;
                private ObjectName    wlmOName   = null;
                private ObjectName    jvmOName   = null;
                private ObjectName    orbtpOName   = null;
                private boolean failed = false;
                private PmiModuleConfig[] configs = null;

            /**
             *  Creates a new test object
             *  (Need a default constructor for the testing framework)
             */
            public PmiJmxTest() {
            }

            /**
             * @param args[0] host
             * @param args[1] port, optional, default is 8880
             * @param args[2] connectorType, optional, default is SOAP connector
             *
             */
            public static void main(String[] args) {
                PmiJmxTest instance = new PmiJmxTest();

                // parse arguments and create AdminClient object
                instance.init(args);

                // navigate all the MBean ObjectNames and cache those we are interested
                instance.getObjectNames();

                // set level, get data, display data
                instance.doTest();

                // test for EJB data
                instance.testEJB();

                // how to use JSR77 getStats method for individual MBean other than PerfMBean
                instance.testJSR77Stats();

            }

            /**
             * parse args and getAdminClient
             */
            public void init(String[] args) {

                try {
                    String  host    = null;
                    String  port    = "8880";
                    String  connector = "SOAP";
                    if(args.length < 1) {
                        System.err.println("ERROR: Usage: PmiJmxTest <host> [<port>]
        [<connector>]");
                        System.exit(2);
                    }
                    else {
                        host = args[0];

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

                        if (args.length > 2)
                            connector = args[2];
                    }

                    if(host == null) {
```

```
                host = "localhost";
            }
            if(port == null) {
                port = "8880";
            }
            if (connector == null) {
                connector = AdminClient.CONNECTOR_TYPE_SOAP;
            }
            System.out.println("host=" + host + " , port=" + port + ",
connector=" + connector);

            //-----------------------------------------------------------------------
            // Get the ac object for the AppServer
            //-----------------------------------------------------------------------
            System.out.println("main: create the adminclient");
            ac = getAdminClient(host, port, connector);

        } catch (Exception ex) {
            failed = true;
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
        }
    }

    /**
     * get AdminClient using the given host, port, and connector
     */
    public AdminClient getAdminClient(String hostStr, String portStr, String connector) {
        System.out.println("getAdminClient: host=" + hostStr + " , portStr=" + portStr);
        AdminClient ac = null;
        java.util.Properties props = new java.util.Properties();
        props.put(AdminClient.CONNECTOR_TYPE, connector);
        props.put(AdminClient.CONNECTOR_HOST, hostStr);
        props.put(AdminClient.CONNECTOR_PORT, portStr);
        try {
            ac = AdminClientFactory.createAdminClient(props);
        }
        catch(Exception ex) {
            failed = true;
            new AdminException(ex).printStackTrace();
            System.out.println("getAdminClient: exception");
        }
        return ac;
    }


    /**
     * get all the ObjectNames.
     */
    public void getObjectNames() {

        try {

            //-----------------------------------------------------------------------
            // Get a list of object names
            //-----------------------------------------------------------------------
            javax.management.ObjectName on = new javax.management.ObjectName("WebSphere:*");

            //-----------------------------------------------------------------------
            // get all objectnames for this server
            //-----------------------------------------------------------------------
            Set objectNameSet= ac.queryNames(on, null);

            //-----------------------------------------------------------------------
            // get the object names that we care about: Perf, Server, JVM,
WLM (only applicable in ND)
            //-----------------------------------------------------------------------
```

```
                    if(objectNameSet != null) {
                        Iterator i = objectNameSet.iterator();
                        while (i.hasNext()) {
                            on = (ObjectName)i.next();
                            String type = on.getKeyProperty("type");

                            // uncomment it if you want to print the ObjectName for each MBean
                            // System.out.println("\n\n" + on.toString());

                            // find the MBeans we are interested
                            if(type != null && type.equals("Perf")) {
                                System.out.println("\nMBean: perf =" + on.toString());
                                perfOName = on;
                            }
                            if(type != null && type.equals("Server")) {
                                System.out.println("\nMBean: Server =" + on.toString());
                                serverOName = on;
                            }
                            if(type != null && type.equals("JVM")) {
                                System.out.println("\nMBean: jvm =" + on.toString());
                                jvmOName = on;
                            }
                            if(type != null && type.equals("WLMAppServer")) {
                                System.out.println("\nmain: WLM =" + on.toString());
                                wlmOName = on;
                            }
                            if(type != null && type.equals("ThreadPool")) {
                                String name = on.getKeyProperty("name");
                                if (name.equals("ORB.thread.pool"))
                                    System.out.println("\nMBean: ORB ThreadPool =" + on.toString());
                                orbtpOName = on;
                            }
                        }
                    }
                    else {
                        System.err.println("main: ERROR: no object names found");
                        System.exit(2);
                    }

                    // You must have Perf MBean in order to get PMI data.
                    if (perfOName == null) {
                        System.err.println("main: cannot get PerfMBean. Make sure PMI is enabled");
                        System.exit(3);
                    }
                }
            catch(Exception ex) {
                failed = true;
                new AdminException(ex).printStackTrace();
                ex.printStackTrace();
            }

    }

    /**
     * Some sample code to set level, get data, and display data.
     */
    public void doTest() {
        try {
            // first get all the configs  - used to set static info for Stats
            // Note: server only returns the value and time info.
            //       No description, unit, etc is returned with PMI data to
reduce communication cost.
            //       You have to call setConfig to bind the static info and Stats data later.
            configs = (PmiModuleConfig[])ac.invoke(perfOName, "getConfigs", null, null);

            // print out all the PMI modules and matching mbean types
            for (int i=0; i<configs.length; i++)
```

```
                System.out.println("config: moduleName=" + configs[i].getShortName() + ",
mbeanType=" + configs[i].getMbeanType());

            // set the instrumentation level for the server
            setInstrumentationLevel(serverOName, null, PmiConstants.LEVEL_HIGH);

            // example to use StatDescriptor.
            // Note WLM module is only available in ND.
            StatDescriptor sd = new StatDescriptor(new String[] {"wlmModule.server"});
            setInstrumentationLevel(wlmOName, sd, PmiConstants.LEVEL_HIGH);

            // example to getInstrumentationLevel
            MBeanLevelSpec[] mlss = getInstrumentationLevel(wlmOName, sd, true);
            // you can call getLevel(), getObjectName(), getStatDescriptor() on mlss[i]

            // get data for the server
            Stats stats = getStatsObject(serverOName, true);
            System.out.println(stats.toString());

            // get data for WLM server submodule
            stats = getStatsObject(wlmOName, sd, true);
            if (stats == null)
                System.out.println("Cannot get Stats for WLM data");
            else
                System.out.println(stats.toString());

            // get data for JVM MBean
            stats = getStatsObject(jvmOName, true);
            processStats(stats);

            // get data for multiple MBeans
            ObjectName[] onames = new ObjectName[]{orbtpOName, jvmOName};
            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);
            // you can call toString or processStats on statsArray[i]

            if (!failed)
                System.out.println("All tests passed");
            else
                System.out.println("Some tests failed");
        }
        catch(Exception ex) {
            new AdminException(ex).printStackTrace();
            ex.printStackTrace();
        }
    }


    /**
     * Sample code to get level
     */
    protected MBeanLevelSpec[] getInstrumentationLevel(ObjectName on, StatDescriptor sd,
boolean recursive) {
        if (sd == null)
            return getInstrumentationLevel(on, recursive);
        System.out.println("\ntest getInstrumentationLevel\n");
        try {
            Object[] params = new Object[2];
            params[0] = new MBeanStatDescriptor(on, sd);
            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);
```

```
                            return mlss;
                }
                catch(Exception e) {
                    new AdminException(e).printStackTrace();
                    System.out.println("getInstrumentationLevel: Exception Thrown");
                    return null;
                }
        }

        /**
         * Sample code to get level
         */
        protected MBeanLevelSpec[] getInstrumentationLevel(ObjectName on, boolean recursive) {
                if (on == null)
                    return null;
                System.out.println("\ntest getInstrumentationLevel\n");
                try {
                    Object[] params = new Object[]{on, new Boolean(recursive)};
                    String[] signature= new String[]{ "javax.management.ObjectName",
"java.lang.Boolean"};
                    MBeanLevelSpec[] mlss = (MBeanLevelSpec[])ac.invoke(perfOName,
"getInstrumentationLevel", params, signature);
                    return mlss;
                }
                catch(Exception e) {
                    new AdminException(e).printStackTrace();
                    failed = true;
                    System.out.println("getInstrumentationLevel: Exception Thrown");
                    return null;
                }
        }

        /**
         * Sample code to set level
         */
        protected void setInstrumentationLevel(ObjectName on, StatDescriptor sd, int level) {
                System.out.println("\ntest setInstrumentationLevel\n");
                try {
                    Object[] params       = new Object[2];
                    String[] signature    = null;
                    MBeanLevelSpec[] mlss = null;
                    params[0] = new MBeanLevelSpec(on, sd, level);
                    params[1] = new Boolean(true);

                    signature= new String[]{ "com.ibm.websphere.pmi.stat.MBeanLevelSpec",
"java.lang.Boolean"};
                    ac.invoke(perfOName, "setInstrumentationLevel", params, signature);
                }
                catch(Exception e) {
                    failed = true;
                    new AdminException(e).printStackTrace();
                    System.out.println("setInstrumentationLevel: FAILED: Exception Thrown");
                }
        }

        /**
         * Sample code to get a Stats object
         */
        public Stats getStatsObject(ObjectName on, StatDescriptor sd, boolean recursive) {

                if (sd == null)
                    return getStatsObject(on, recursive);

                System.out.println("\ntest getStatsObject\n");
                try {
                    Object[] params    = new Object[2];
                    params[0] = new MBeanStatDescriptor(on, sd);  // construct MBeanStatDescriptor
```

```
            params[1] = new Boolean(recursive);
            String[] signature = new String[] {
"com.ibm.websphere.pmi.stat.MBeanStatDescriptor", "java.lang.Boolean"};
            Stats stats  = (Stats)ac.invoke(perfOName, "getStatsObject", params, signature);

            if (stats == null) return null;

            // find the PmiModuleConfig and bind it with the data
            String type = on.getKeyProperty("type");
            if (type.equals(MBeanTypeList.SERVER_MBEAN))
                setServerConfig(stats);
            else
                stats.setConfig(findConfig(on));

            return stats;

        } catch(Exception e) {
            failed = true;
            new AdminException(e).printStackTrace();
            System.out.println("getStatsObject: Exception Thrown");
            return null;
        }
    }

    /**
     * Sample code to get a Stats object
     */
    public Stats getStatsObject(ObjectName on, boolean recursive) {
        if (on == null)
            return null;

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

        try {
            Object[] params  = new Object[]{on, new Boolean(recursive)};
            String[] signature = new String[] { "javax.management.ObjectName",
"java.lang.Boolean"};
            Stats stats  = (Stats)ac.invoke(perfOName, "getStatsObject", params,
signature);

            // find the PmiModuleConfig and bind it with the data
            String type = on.getKeyProperty("type");
            if (type.equals(MBeanTypeList.SERVER_MBEAN))
                setServerConfig(stats);
            else
                stats.setConfig(findConfig(on));

            return stats;

        } catch(Exception e) {
            failed = true;
            new AdminException(e).printStackTrace();
            System.out.println("getStatsObject: Exception Thrown");
            return null;
        }
    }

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

    /**
     * Sample code to navigate and get the data value from the Stats and Statistic object.
     */
```

```
private 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);

    // list 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("");

    // 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
    // Also show how to do translation.
    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())
                            + ", 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()
                                + ", integral=" + data.getIntegral()
                                + ", avg=" + data.getMean()
                                + ", lowWaterMark=" + data.getLowWaterMark()
                                + ", highWaterMark=" + data.getHighWaterMark());
            }
        }
    }

    // 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 + "    ");
    }
}

/**
 * Get PmiModuleConfig based on MBean ObjectName
 */
public PmiModuleConfig findConfig(ObjectName on) {
    if (on == null) return null;
```

```
            String type = on.getKeyProperty("type");
            System.out.println("findConfig: mbean type =" + type);

            for (int i=0; i<configs.length ; i++) {

                if (configs[i].getMbeanType().equals(type))
                    return configs[i];
            }
            System.out.println("Error: cannot find the config");
            return null;

    }

    /**
     * Get PmiModuleConfig based on PMI module name */
    public PmiModuleConfig findConfig(String moduleName) {
        if (moduleName == null) return null;

        for (int i=0; i<configs.length ; i++) {

            if (configs[i].getShortName().equals(moduleName))
                return configs[i];
        }
        System.out.println("Error: cannot find the config");
        return null;

    }

    /**
     * The Stats object returned from server does not have static config info.
You have to set it on client side.
     */
    public void setServerConfig(Stats stats) {
        if(stats == null) return;
        if(stats.getType() != TYPE_SERVER) return;

        PmiModuleConfig config = null;

        Stats[] statList = stats.getSubStats();
        if (statList == null || statList.length == 0)
            return;
        Stats oneStat = null;
        for(int i=0; i<statList.length; i++) {
            oneStat = statList[i];
            if (oneStat == null) continue;
            config = findConfig(oneStat.getName());
            if(config != null)
                oneStat.setConfig(config);
            else
                System.out.println("Error: get null config for " + oneStat.getName());
        }
    }

    /**
     * sample code to show how to get a specific MBeanStatDescriptor
     */
    public MBeanStatDescriptor getStatDescriptor(ObjectName oName, String name) {
        try {
            Object[] params = new Object[]{serverOName};
            String[] signature= new String[]{"javax.management.ObjectName"};
            MBeanStatDescriptor[] msds = (MBeanStatDescriptor[])ac.invoke(perfOName,
"listStatMembers", params, signature);
            if (msds == null)
                return null;
            for (int i=0; i<msds.length; i++) {
                if (msds[i].getName().equals(name))
                    return msds[i];
```

```
                }
                return null;
            }
            catch(Exception e) {
                new AdminException(e).printStackTrace();
                System.out.println("listStatMembers: Exception Thrown");
                return null;
            }

        }

        /**
         * sample code to show you how to navigate MBeanStatDescriptor via listStatMembers
         */
        public MBeanStatDescriptor[] listStatMembers(ObjectName mName) {
            if (mName == null)
                return null;

            try {
                Object[] params = new Object[]{mName};
                String[] signature= new String[]{"javax.management.ObjectName"};
                MBeanStatDescriptor[] msds = (MBeanStatDescriptor[])ac.invoke(perfOName,
"listStatMembers", params, signature);
                if (msds == null)
                    return null;
                for (int i=0; i<msds.length; i++) {
                    MBeanStatDescriptor[] msds2 = listStatMembers(msds[i]);
                }
                return null;
            }
            catch(Exception e) {
                new AdminException(e).printStackTrace();
                System.out.println("listStatMembers: Exception Thrown");
                return null;
            }

        }


        /**
         * Sample code to get MBeanStatDescriptors
         */
        public MBeanStatDescriptor[] listStatMembers(MBeanStatDescriptor mName) {
            if (mName == null)
                return null;

            try {
                Object[] params = new Object[]{mName};
                String[] signature= new String[]{"com.ibm.websphere.pmi.stat.MBeanStatDescriptor"};
                MBeanStatDescriptor[] msds = (MBeanStatDescriptor[])ac.invoke(perfOName,
"listStatMembers", params, signature);
                if (msds == null)
                    return null;
                for (int i=0; i<msds.length; i++) {
                    MBeanStatDescriptor[] msds2 = listStatMembers(msds[i]);
                    // you may recursively call listStatMembers until find the one you want
                }
                return msds;
            }
            catch(Exception e) {
                new AdminException(e).printStackTrace();
                System.out.println("listStatMembers: Exception Thrown");
                return null;
            }

        }
```

```
    /**
     * sample code to get PMI data from beanModule
     */
    public void testEJB() {

        // This is the MBeanStatDescriptor for Enterprise EJB
        MBeanStatDescriptor beanMsd = getStatDescriptor(serverOName, PmiConstants.BEAN_MODULE);
        if (beanMsd == null)
            System.out.println("Error: cannot find beanModule");

        // get the Stats for module level only since recursive is false
        Stats stats = getStatsObject(beanMsd.getObjectName(), beanMsd.getStatDescriptor(),
false); // pass true if you wannt data from individual beans

        // find the avg method RT
        TimeStatisticImpl rt = (TimeStatisticImpl)stats.getStatistic(EJBStatsImpl.METHOD_RT);
        System.out.println("rt is " + rt.getMean());

        try {
            java.lang.Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        // get the  Stats again
        Stats stats2 = getStatsObject(beanMsd.getObjectName(), beanMsd.getStatDescriptor(),
false); // pass true if you wannt data from individual beans

        // find the avg method RT
        TimeStatisticImpl rt2 = (TimeStatisticImpl)stats2.getStatistic(EJBStatsImpl.METHOD_RT);
        System.out.println("rt2 is " + rt2.getMean());

        // calculate the difference between this time and last time.
        TimeStatisticImpl deltaRt = (TimeStatisticImpl)rt2.delta(rt);
        System.out.println("deltaRt is " + rt.getMean());

    }

    /**
     * Sample code to show how to call getStats on StatisticProvider MBean directly.
     */
    public void testJSR77Stats() {
        // first, find the MBean ObjectName you are interested.
        // Refer method getObjectNames for sample code.

        // assume we want to call getStats on JVM MBean to get statistics
        try {

            com.ibm.websphere.management.statistics.JVMStats stats =
                    (com.ibm.websphere.management.statistics.JVMStats)ac.invoke(jvmOName,
"getStats", null, null);

            System.out.println("\n get data from JVM MBean");

            if (stats == null) {
                System.out.println("WARNING: getStats on JVM MBean returns null");
            } else {

                // first, link with the static info if you care
                ((Stats)stats).setConfig(findConfig(jvmOName));

                // print out all the data if you want
                //System.out.println(stats.toString());

                // navigate and get the data in the stats object
                processStats((Stats)stats);
```

```
                    // call JSR77 methods on JVMStats to get the related data
                    com.ibm.websphere.management.statistics.CountStatistic upTime =
stats.getUpTime();
                    com.ibm.websphere.management.statistics.BoundedRangeStatistic heapSize =
stats.getHeapSize();

                    if (upTime != null)
                        System.out.println("\nJVM up time is " + upTime.getCount());
                    if (heapSize != null)
                        System.out.println("\nheapSize is " + heapSize.getCurrent());
                }
            } catch (Exception ex) {
                ex.printStackTrace();
                new AdminException(ex).printStackTrace();
            }
        }
    }
}
```

Developing Performance Monitoring Infrastructure interfaces (Version 4.0)

Before you begin

The Version 4.0 APIs are supported in this release, however, some data hierarchy changes have occurred in the PMI modules, including the enterprise bean and HTTP sessions modules. If you have an existing PmiClient application and you want to run it against Version 5.0, you might have to update the PerfDescriptor(s) based on the new PMI data hierarchy.

The getDataName and getDataId methods in PmiClient have also changed. They are now non-static methods in order to support multiple WebSphere Application Server versions. You might have to update your existing application which uses these two methods.

This section discusses the use of the Performance Monitoring Infrastructure (PMI) client interfaces in applications. The basic steps in the programming model follow:

Steps for this task

1. Retrieve an initial collection or snapshot of performance data from the server.

   A client uses the CpdCollection interface to retrieve an initial collection or snapshot from the server. This snapshot, which is called Snapshot in this example, is provided in a hierarchical structure as described in data organization and hierarchy, and contains the current values of all performance data collected by the server. The snapshot maintains the same structure throughout the lifetime of the CpdCollection instance.

2. Process and display the data as specified.

   The client processes and displays the data as specified. Processing and display objects, for example, filters and GUIs, can register as CpdEvent listeners to data of interest. The listener works only within the same Java virtual machine (JVM). When the client receives updated data, all listeners are notified.

3. Display the new CpdCollection instance through the hierarchy.

   When the client receives new or changed data, the client can simply display the new CpdCollection instance through its hierarchy. When it is necessary to update the Snapshot collection, the client can use the update method to update Snapshot with the new data.

   ```
   Snapshot.update(S1);
   // ...later...
   Snapshot.update(S2);
   ```

Steps 2 and 3 are repeated through the lifetime of the client.

Third-party performance monitoring and management solutions

Several other companies provide performance monitoring, problem determination and management solutions that can be used with WebSphere Application Server.

These products use WebSphere Application Server interfaces, including Performance Monitoring Infrastructure (PMI), Java Management Extensions (JMX), and PMI Request Metrics Application Response Measurement (ARM).

See the topic Performance: Resources for learning for a link to IBM business partners providing monitoring solutions for WebSphere Application Server.

Measuring data requests (Performance Monitoring Infrastructure Request Metrics)

Performance Monitoring Infrastructure (PMI) Request Metrics collects data by timing requests as they travel through WebSphere Application Server components. This data helps to identify run time and application problems. PMI Request Metrics logs the time spent at major points, such as the Web server plug-in, Web container, enterprise bean container and database. These points are recorded in logs and can be written to Application Response Measurement (ARM) agents used by Tivoli monitoring tools. Request metrics provides response times for remote EJB containers (by way of IIOP), , servlets (Web containers) and Java Database Connectivity (JDBC) drivers.

If you plan to run in a production environment, plan to filter by IP address - a specific IP address using a synthetic transaction generator. If you choose to enable request metrics, but not filter by a specific IP address, performance can be impacted significantly.

Learn more about Request Metrics by reviewing this section, including:
* Detailed explanation about Request Metrics
* Request Metrics process and filters
* Types and format of output you will be reading
* Configuring Request Metrics

Performance Monitoring Infrastructure Request Metrics

Performance Monitoring Infrastructure (PMI) Request Metrics helps identify run time and application performance problems by capturing process hop response times in multi-tiered applications and recording the data in system logs.

For requests that start from either an HTTP or enterprise bean remote requests, Request Metrics captures 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.

When active, Request Metrics compares each incoming request to a set of known 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 fan out across several nodes within a distributed system. Each process can further fan out and call other processes. When the processes fan out, trace records are generated for each process. Then, 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.

Application Response Measurement

Application Response Measurement (ARM) is an Open Group standard composed of a set of interfaces implemented by an ARM agent that provides information on elapsed time for process hops.

WebSphere Application Server does not provide an ARM agent. Contact your ARM agent provider for information on whether their ARM agent is supported with WebSphere Application Server.

See the article Performance: Resources for learning for more information about the ARM specifications.

Performance Monitoring Infrastructure Request Metrics trace filters

When Performance Monitoring Infrastructure (PMI) Request Metrics is active, trace filters control which requests get traced. The data is recorded to the system log file StdOut and can be used for real-time and historical analysis.

Incoming HTTP requests

For HTTP requests arriving at a WebSphere Application Server it is possible to filter on the URI and client IP address.

- **Client IP address filters.** Requests are filtered based on a known IP address. You can specify a mask for an IP address using the asterisk (*). If used, the asterisk must always be the last character of the mask, for example 127.0.0.*, 127.0.*, 127*. 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. The rules for pattern matching are the same as for matching client IP address filters.
- **Filter combinations.** If both URI and Client IP address filters are active, then Request Metrics requires a match for both filter types. If neither is active, all requests are considered a match.

Incoming enterprise bean requests

- **Enterprise bean method name filters.** Requests are filtered 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.

Performance Monitoring Infrastructure Request Metrics data output

The trace records for Performance Monitoring Infrastructure (PMI) Request Metrics data are written in two log files

In the WebSphere Application Server log file the trace record format is:

```
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
```

The trace record format is composed 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 application server log files and the `PLUGIN` trace record from the Web server plug-in log file. 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 separating fields following the keyword "parent:". Likewise, the current correlator is denoted by the comma separating fields following "current:".

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

Following the parent and current correlators, is the metrics data for timed operation:
- **type**: A code representing the type of operation being timed. Supported types include , URI, EJB and JDBC.
- **detail**: Identifies the name of the operation being timed (See the description of Universal Resource Identifier (URI), , Enterprise bean and Java Database Connectivity (JDBC) below.)
- **elapsed**: The measured elapsed time in <units> for this operation, which includes all sub-operations called by this operation. The unit of elapsed time is milliseconds.

The type and detail fields are described as follows:
- `URI`: The trace record was generated by a Web component. The URI is the name of the URI used to invoke the request.
- `EJB`: 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.

Configuring Request Metrics

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

To configure Request Metrics, you will need to access the Configuration tab in the administrative console. To access the Configuration tab , click **Problem Determination** > **PMI Request Metrics** from the administrative console navigation tree.

Tasks included in configuring Request Metrics:

Steps for this task
1. Enable Request Metrics.
2. **(Optional)** Enable Application Response Measurement (ARM).
3. **(Optional)** Enable Request Metrics filters.
4. **(Optional)** Add and remove Request Metrics filters.
5. Set the trace level in Request Metrics.

Enabling Performance Monitoring Infrastructure Request Metrics

When enabled, Performance Monitoring Infrastructure (PMI) Request Metrics captures response times for the initiating request and any related downstream enterprise bean invocations and Java Database Connectivity (JDBC) calls. Then, Request Metrics compares each incoming request to a set of known filters.

Steps for this task
1. Open the administrative console.
2. Click **Problem Determination** > **PMI Request Metrics** in the console navigation tree.
3. Select the check box in the **enable** field under the Configuration tab.
4. Click **Apply** or **OK**.
5. Click **Save**.

What to do next

Regenerate the Web server plug-in configuration file, if logging time spent in the Web server.

Enabling Application Response Measurement

Before you begin

Before enabling Application Response Measurement (ARM), install an appropriate ARM implementation on all WebSphere Application Server nodes. Refer to the appropriate ARM implementation documentation. Verify with your ARM agent provider that Request Metrics is supported by the ARM agent implementation. ARM support is dependent on Request Metrics support.

You can learn more about ARM agents in the topic Performance: Resources for Learning.

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

Steps for this task

1. Install the appropriate ARM implementation

   a. Change the startup command for the application servers to include the following:

      `-Dcom.ibm.websphere.pmi.reqmetrics.ARMIMPL=`*`ARMIMPLNAME`*

      ARM support 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. Open the administrative console.
3. Click **Problem Determination** > **PMI Request Metrics** in the console navigation tree.
4. Select the check box in the **enableARM** field.
5. Click **Apply** or **OK**.
6. Click **Save**.

What to do next

Regenerate the Web server plug-in configuration file.

Enabling Performance Monitoring Infrastructure Request Metrics filters

Performance Monitoring Infrastructure (PMI) Request Metrics compares each incoming request to a set of known filters, but you need to enable these filters.

Steps for this task

1. Open the administrative console.
2. Click **Problem Determination** > **PMI Request Metrics** in the administrative console navigation tree.
3. Click **filters**.
4. Click *filter type*.
5. Select the check box in the **enable** field under the Configuration tab.
6. Click **Apply** or **OK**.
7. Click **Save**.

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

What to do next

Regenerate the Web server plug-in configuration file, if logging time spent in the Web server.

Adding and removing Performance Monitoring Infrastructure Request Metrics filters

To add or remove Performance Monitoring Infrastructure (PMI) Request Metrics filters:

Steps for this task

1. Open the administrative console.
2. Click **Problem Determination** > **PMI Request Metrics** in the console navigation tree.
3. Click **filters**.
4. Click **New**.
5. Choose a filter type from the drop down box in the **type** field under the Configuration tab.
6. (Optional) Select the check box in the **enable** field to enable the filter.
7. Click **Apply** or **OK**.
8. Click **Save**.

   Individual filters are composed of an indicator and an IP address. Use the indicator to determine whether the individual filter is active. The IP address is composed of a standard dotted IP address.

What to do next

Regenerate the Web server plug-in configuration file, if logging time spent in the Web server.

Setting the trace level in Performance Monitoring Infrastructure Request Metrics

To set the trace level to generate records:

Steps for this task

1. Open the administrative console.
2. Click **Problem Determination** > **PMI Request Metrics** in the administrative console navigation tree.
3. Find **traceLevel** in the Configuration tab.
4. Select the desired trace level from the drop down list box.

   To set the Request Metrics trace level to generate records, make sure the trace level is set to a value greater than NONE.
5. Click **Apply** or **OK**.
6. Click **Save**.

What to do next

Regenerate the Web server plug-in configuration file, if logging time spent in the Web server.

Performance Monitoring Infrastructure Request Metrics

Use this page to enable Performance Monitoring Infrastructure (PMI) Request Metrics, enable Request Metrics Application Response Measurement (ARM), and set trace levels.

To view this administrative console page, click **Troubleshooting** > **PMI Request Metrics**.

Request Metrics: Enables PMI Request Metrics.

When disabled, the Request Metrics function is disabled.

Application Response Measurement (ARM): Enables PMI Request Metrics to call an underlying ARM agent.

Before enabling ARM, install an appropriate ARM implementation on all WebSphere Application Server nodes. Verify with your ARM agent provider that Request Metrics is supported by the ARM agent implementation. ARM support is dependent on Request Metrics support.

Trace Level: Specifies how much trace data to accumulate for a given request.

Including one of the following: NONE - no trace; HOPS - only accumulates on major process hops; PERF_DEBUG - enables additional information over hops, but is not as performance intensive as DEBUG; DEBUG - full detailed trace.

PMIRM Filter collection
Use this page to view a list of Performance Monitoring Infrastructure (PMI) Request Metrics filters.

To view this administrative console page, click **Troubleshooting** > **PMI Request Metrics** > **Filters**.

Type: Specifies the type of request metrics filter.

Enable: Specifies whether this filter is enabled.

PMIRM Filter settings
Use this page to specify filters that define whether or not trace is enabled for the request as it moves through WebSphere Application Server.

To view this administrative console page, click **Troubleshooting** > **PMI Request Metrics** > **filters** > *filter_type*.

Type: Specifies the type of Request Metrics filter.

Enable: Specifies whether this filter is enabled.

filterValues collection
Use this page to specify the values for client IP, URI or EJB Request Metrics filters.

To view this administrative console page, click **Troubleshooting** > **PMI Request Metrics** > **filters** > *filter_type* > **filterValues**.

Value: Specifies a URI value or IP name based on the type of filter.

For example, for URI filters, the value might be "/servlet/snoop".

Enable Filter: Specifies whether a filter value is enabled.

filterValues settings
Use this page to specify the values for client IP, URI or EJB Request Metrics filters.

To view this administrative console page, click **Troubleshooting** > **PMI Request Metrics** > **filters** > *filter* > **filterValues** > *filter_value*.

Value: Specifies a URI value or IP name based on the type of filter.

For example, for URI filters, the value can be "/servlet/snoop".

Enable: Specifies whether this filter value is enabled.

Regenerating the Web server plug-in configuration file

After you modify the Request Metrics configuration, you must complete the following steps to regenerate the Web server plug-in configuration file. Regenerating ensures that the Web server plug-in recognizes the changes you made for the Request Metrics configuration. If you are making multiple changes to Request Metrics, then regenerate the plug-in configuration files once you have completed all changes.

Note: You must complete this step after you change the request metrics configuration. If you do not, the Web server plug-in might have different Request Metrics configuration data than 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.

Steps for this task
1. Open the administrative console.
2. (Regenerate the Web server plug-in configuration).

Example: Generating trace records from Performance Monitoring Infrastructure Request Metrics

Examples

Use `HitCount` enterprise bean `/webapp/examples/hitcount?source=EJB` where the servlet is deployed on one machine - 192.168.0.1, and the enterprise bean `Increment.jar` file is deployed on a second machine - 192.168.0.2. The web server runs on 192.168.0.1.

In this example, both machines are used as clients.

To illustrate the use of client IP filtering, one client IP filter (192.168.0.2) is defined and enabled. This action allows tracing of requests originating from the enterprise bean machine through `http://192.168.0.1/webapp/examples/hitcount?source=EJB`. However, requests originating from the servlet machine are not traced since the client IP address is not in the filter list.

By only creating a client IP filter, any request from that client IP address is effectively traced. This tool can be effective for locating performance problems with systems under load. If the normal load is originating from other IP addresses, then their requests are not traced. By using the defined client IP address to generate requests, you can see performance bottlenecks at the various hops by comparing the trace records of the loaded system to trace records from a non-loaded run. This ability can help focus tuning efforts to the correct node and process within a complex deployment environment.

Make sure Request Metrics is enabled using the administrative console. Also, make sure the trace level is set to at least `hops` (writing request traces at process boundaries). Using the configuration listed above, send a request through the `HitCount` servlet from the enterprise bean machine `http://192.168.0.1/webapp/examples/hitcount?source=EJB`.

In this example, at least three trace records are generated:
- A trace record for the Web server plug-in appears in the plug-in log file on 192.168.0.1.

- A trace record for the servlet execution appears in the application server log file on 192.168.0.1.
- A trace record for the increment bean method invocation appears in the application server log file on 192.168.0.2

The two trace records appearing on 192.168.0.1 are similar to the following:

```
PLUGIN: parent:ver=1,ip=192.168.0.1,time=1016556185102,pid=796,reqid=40,event=0
- current:ver=1,ip=192.168.0.1,time=1016556185102,pid=796,reqid=40,event=1
type=HTTP detail=/webapp/examples/hitcount elapsed=60 bytesIn=0 bytesOut=2252

PMRM0003I: parent:ver=1,ip=192.168.0.1,time=1016556185102,pid=796,reqid=40,event=0
- current:ver=1,ip=192.168.0.1,time=1016556186102,pid=884,reqid=40,event=1
type=URI detail=/webapp/examples/hitcount elapsed=60
```

The trace record appearing on 192.168.0.2 is similar to the following:

```
PMRM0003I: parent:ver=1,ip=192.168.0.1,time=1016556186102,pid=884,reqid=40,event=1
- current:ver=1,ip=192.168.0.2,time=1016556122505,pid=9321,reqid=40,event=1
type=EJB detail=com.ibm.websphere.examples.Inc.IncBean.increment elapsed=40
```

Performance monitoring service settings

Use this page to specify settings for performance monitoring, including enabling performance monitoring, selecting the PMI module and setting monitoring levels.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Performance Monitoring**.

Startup

Specifies whether the application server attempts to start the specified service. If an application server is started when the performance monitoring service is disabled, you will have to restart the server in order to enable it.

Initial specification level

Specifies a Performance Monitoring Infrastructure (PMI) string that stores PMI specification levels, for example module levels, for all components in the server.

Set the PMI specification levels by selecting the *none*, *standard* or *custom* checkbox. If you choose *none*, all PMI modules are set to the *none* level. Choosing *standard*, sets all PMI modules to *high* and enables all PMI data excluding the method level data and JVMPI data. Choosing custom, gives you the option to change the level for each individual PMI module. You can set the level to N, L, M, H or X (none, low, medium, high and maximum). **Note** that you should not change the module names.

Specifications

Specifies the PMI module and monitoring level that you have set.

Set the PMI specification levels by selecting the *none*, *standard* or *custom* checkbox. If you choose *none*, all PMI modules are set to the *none* level. Choosing *standard*, sets all PMI modules to *high* and enables all PMI data excluding the method level data and JVMPI data. Choosing custom, gives you the option to change the level for each individual PMI module. You can set the level to N, L, M, H or X (none, low, medium, high and maximum). **Note** that you should not change the module names.

Performance: Resources for learning

Use the following links to find relevant supplemental information about performance. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas. The following sections are covered in this reference:

View links to additional information about:

- Performance Monitoring Infrastructure (PMI) Request Metrics.
- Monitoring performance with third-party tools
- Tuning performance
- Garbage collection

Performance Monitoring Infrastructure (PMI) Request Metrics

- **Systems Management: Application Response Measurement (ARM)** (http://www.opengroup.org/publications/catalog/c807.htm)

  The Open Group ARM specifications.

Monitoring performance with third-party tools

- **Enterprise Web Application Management** http://www-3.ibm.com/software/webservers/pw/dhtml/wsperformance/... performance_bpsolutions.html WebSphere Performance Management Business Partner Solution Finder

  Find a list of IBM's business partners that offer performance monitoring tools compliant with WebSphere Application Server.

Tuning performance

- http://www-3.ibm.com/software/webservers/httpservers/doc/v136/misc/perf.html Hints on Running a high-performance Web server

  Read hints about running Apache on a heavily loaded Web server. The suggestions include how to tune your kernel for the heavier TCP/IP load, and hardware and software conflicts

- **Application tuning** http://www-1.ibm.com/support/docview.wss?uid=swg27000615

  See WebSphere Application Server Development Best Practices for Performance and Scalability for more information on application tuning.

- **Performance Analysis for Java Web sites** http://www.awprofessional.com/catalog/product.asp?product_id={A801214C-A166-4836-859A-423B246C65E4}

- **AIX documentation** http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base/aix.htm

  View the entire AIX software documentation library for releases 4.3, 5.1, and 5.2.

- **WebSphere Application Server Development Best Practices for Performance and Scalability** http://www-1.ibm.com/support/docview.wss?uid=swg27000615

Describes development best practices for Web applications with servlets, JSP files, JDBC connections, and enterprise applications with EJB components.

- iSeries performance documents http://www-1.ibm.com/servers/eserver/iseries/software/websphere/wsappserver/product/...performanceAE40.html

  This Web site is a directory to several iSeries performance documents, including WebSphere Application Server for iSeries Performance Considerations, the Performance Trace Data Visualizer (PTDV) tool and Workload Estimator tool.

- **IBM WebSphere Application Server Advanced Edition Tuning Guide** http://www-3.ibm.com/software/webservers/appserv/doc/v40/ae/infocenter/was/pdf/...nav_Tuneguide.pdf (Version 4.02)

- **Redbook: WebSphere Application Server V3.5 Handbook (SG24-6161-00)** http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246161.pdf

- **Redbook: WebSphere Application Server V3 Performance Tuning Guide (SG24-5657-00)**
  http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245657.pdf

Garbage collection

- **IBM developerWorks** http://java.sun.com/docs/hotspot/gc/index.html

  Search the IBM developerWorks Web site for a list of garbage collection documentation, including "Understanding the IBM Java Garbage Collector", a three-part series. To locate the documentation, search on "sensible garbage collection" in the developerWorks search application.

  Review "Understanding the IBM Java Garbage Collector" for a description of the IBM verbose:gc output and more information about the IBM garbage collector.

- **Tuning Garbage Collection with the 1.3.1 JavaTM Virtual Machine** http://java.sun.com/docs/hotspot/gc/index.html

  Learn more about using garbage collection in a Solaris operating environment.

Chapter 3. Tuning performance

To optimize your WebSphere Application Servers to their fullest extent, use the Performance Advisors in addition to the suggested procedures or parameters in the tuning parameter hot list and the tuning parameter index.

Performance Advisors

The Performance Advisors use the PMI data to suggest configuration changes to ORB service thread pools, Web container thread pools, connection pool size, persisted session size and time, prepared statement cache size, and session cache size. The Runtime Performance Advisor runs in the application server process, while the other advisor runs in the Tivoli Performance Viewer (TPV). For more information, see Using the Runtime Performance Advisor and Using the Performance Advisor in Tivoli Performance Viewer.

Tuning Parameter hot list and parameter index

Review the Tuning parameter hot list, which is a subset of the tuning parameter index. These hot parameters have an important impact on performance.

For an index of the tuning parameters used in the WebSphere product, see the Tuning parameter index. Use this index to find information and parameters about general tuning recommendations. You can tune:
- Application servers
- Java virtual machines
- Applications
- Databases
- Java Messaging Service
- Security
- Hardware
- Operating systems
- Web servers

The tuning guide focuses on server tuning. If you want to tune your applications, see Performance: Resources for learning for more information about application tuning.

For your convenience, procedures for tuning parameters in other products, such as DB2, Web servers and operating systems are included. Because these products might change, consider these descriptions as suggestions.

Tuning parameter hot list

It is recommended that you first review the following tuning parameter hot list, which is a subset of the tuning parameter index. These parameters are on a hot list because they have an important impact on performance. Because they are application dependent, the appropriate parameter settings for specific applications and environments can vary.
- Hardware and capacity settings

- Java virtual machine heap size
- Application assembly performance checklist
- Data sources connection pool and prepared statement cache
- Solaris operating system TCP_TIME_WAIT_INTERVAL
- Pass by value/Pass by reference
- IBM HTTP Server access logs
- HTTP keep alive connections
- Transaction logs

Tuning parameter index

Each WebSphere Application Server process has several parameters influencing 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.

It is recommended that you first review the Tuning parameter hot list, which is a subset of the tuning parameter index. These parameters have an important impact on performance. Because these parameters are application dependent, the parameter settings for specific applications and environments can vary.

Each parameter in the tuning parameter index links to information that explains the parameter, provides reasons to adjust the parameter, how to view or set the parameter, as well as default and recommended values.

- Application servers
- Java virtual machines
- Applications
- Databases
- Java messaging service
- Security
- Hardware capacity and settings
- Operating systems
- Web servers

Application servers

By tuning application server settings, you can control how an application server provides services for running applications and their components. WebSphere Application Server contains interrelated components that must be harmoniously tuned to support the custom needs of your end-to-end e-business application. This group of interrelated components is known as the queuing network. The queuing network helps the system achieve maximum throughput while maintaining the overall stability of the system.

You can tune the following application server settings:
- **Object Request Broker**
- **XML parser definitions**
- **Dynamic cache service**
- **Web container**
- **EJB container**

- **Session management**
- **Data sources**

Java virtual machines

The application server, being a Java process, requires a Java virtual machine (JVM) to run, and to support the Java applications running on it. As part of configuring an application server, you can fine-tune settings that enhance system use of the JVM. In addition to the following tuning parameters, see also Java memory tuning tips.

Use the following JVM parameters, including garbage collection options for IBM Developer Kit 1.3.1, to tune the Java virtual machine.

- **Generic JVM arguments**
  - Quickstart (-Xquickstart)
  - Avoiding class verification (-Xverify:none)
  - Class garbage collection (-Xnoclassgc)
  - Garbage collection threads (-Xgcthreads)
  - Heap compaction (-Xnocompactgc)
  - Initial system heap size (-Xinitsh)
  - Thread local heap size (-Xmc)
  - Local cache object allocation (-Xml)
- Initial heap size
- Maximum heap size
- Disable just in time (JIT) compiler
- **Garbage collection policy**
  - **Description:** Specifies if the garbage collection policy (gcpolicy) is set to `optavgpause` so that concurrent marking is used to track application threads starting from the stack before the heap becomes full. The garbage collector pauses become uniform and long pauses are not apparent. The trade-off is reduced throughput because threads might have to do extra work.
  - **How to view or set:**
    1. Open the administrative console.
    2. Click **Servers** > **Application Servers** > *server_name*.
    3. Click the **Process Definition** > **Java Virtual Machine** under Additional Properties.
    4. Enter the value `-Xgcpolicy:[optthruput|optavgpause]` in the **Generic JVM Arguments** field.
    5. Click **Apply** or **OK**.
  - **Default value:** `optthruput`
  - **Recommended value:** `optthruput`
- **Sun JDK 1.3 Generational Garbage Collection**
  - **Description:** The Sun JDK 1.3 introduces generation garbage collection, which makes use of separate memory pools to contain objects of different ages. GC cycle collects these objects independently from one another. Additional parameters allows to individually set the size of these pools. To achieve better performance, set size of pool containing short lived objects sufficient enough, so that objects in the pool do not live through more then one garbage collection cycle. Size of new generation pool is determined by NewSize & MaxNewSize parameters. Objects that survive first GC cycle gets transferred

to another pool. Size of that pool is determined by parameter SurvivorRatio. If, garbage collection becomes a bottleneck, try customizing the generation pool settings.

– **How to view or set:**
  1. Open the administrative console.
  2. Click **Servers** > **Application Servers** > *server_name*.
  3. Click the **Process Definition** > **Java Virtual Machine** under Additional Properties.
  4. Click **Custom Properties** under Additional Properties.
  5. Enter the following values in the **Generic JVM Arguments** field: `-XX:NewSize (lower bound)` , `-XX:MaxNewSize (upper bound)`, and `-XX:SurvivorRatio=NewRatioSize`.
  6. Click **Apply** or **OK**.
  7. Click **Save** on the console taskbar.
  8. Stop and restart the application server.

  You can monitor garbage collection statistics using object statistics in the Tivoli Performance Viewer or the verbose:gc configuration setting.

– **Default value:** `NewSize=2m`, `MaxNewSize=32m`, `SurvivorRartio=2`

– **Recommended value:** `-XX:newSize=640m` `-XX:MaxNewSize=640m` `-XX:SurvivorRatio=16` (for JVM with more then 1 GB heap size) or set 50 to 60% of total heap size to new generation pool.

– **Additional Information:** More information about generational garbage collection can be found at http://java.sun.com/docs/hotspot/gc/index.html.

- **Sun JDK 1.3 HotSpot -server warm-up**
  – **Description:** Introduces adaptive JVM technology containing algorithms for optimizing byte code execution over time. The JVM runs in two modes, -server and -client. Performance is significantly enhanced if running in -server mode when a sufficient amount of time is allowed for a HotSpot JVM to warm up by performing continuous execution of byte code. In most cases, run the -server mode. This produces more efficient run-time execution over extended periods. Use the -client mode if you prefer a faster startup time and a smaller memory footprint, at the cost of lower extended performance.
  – **How to view or set:** (Java virtual machine page) in the administrative console to change the -client or -server mode. You can monitor the process size and the server startup time to check the difference between -client and -server.
  – **Default value:** -server
  – **Recommended value:** -server

Applications

Several topics including Web modules, EJB modules, client modules, Web services and application services comprise the application programming model and provide numerous services supporting deployed applications. The Application Assembly Tool performance checklist introduces important topics for tuning during application assembly.

Databases

Tuning parameters vary according to the type of database you are using. DB2 tuning parameters are provided for your convenience. Since DB2 is not a WebSphere Application Server product and can change, consider these descriptions as suggestions.

Java messaging service

You can tune Java messaging service (JMS) run-time components, resources, and the embedded messaging server through the administrative console with the following parameters:

- **Listener service**
  - Thread pool
  - Custom properties Application Server Facilities and Non-Application Server Facilities
  - Listener port Maximum sessions
  - Listener port Maximum messages
- **JMS resources**
  - XA enabled
  - Connection pool size
  - WebSphere MQ queue connection factory Transport type
  - WebSphere MQ topic connection factory Transport type
  - Transaction log directory
  - **WebSphere MQ**
    - **Description:** Includes the WebSphere MQ folder where WebSphere MQ is installed and indirectly where its data files reside since they are always under the directory *WebSphereMQHome/data*. Use this parameter during installation when extra disks are available.
    - **How to view or set:** To view, search your system for a folder named WebSphere MQ. Set during installation.
    - **Default value:** *c:\Program Files\IBM\WebSphere MQ*
    - **Recommended value:** Locate this directory away from any other disk input or output, on the fastest disk available. Redundant Array of Inexpensive Disks (RAID) is recommended.
  - **Log file pages**
    - **Description:** Controls the size of the queue manager log files in number of pages. Use this parameter when high volumes of messages are being sent through a server.
    - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **QueueManager** > *QM_name* > **Log** > **LogFilePages**. For embedded JMS, set the LogFilePages to the desired value. Run the **deletemq** and **createmq** commands in the *install_root*\bin directory to delete or recreate the queue manager. For external JMS, set LogFilePages to the desired value. Do this before creating the queue manager in the registry at **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **Log Defaults**. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
    - **Default value:** External: 256, embedded: 512
    - **Recommended value:** There is no performance penalty for setting this value too high. Therefore, set the value to its maximum of 16384 pages.

– **Log buffer pages**
  - **Description:** Controls the size of the queue manager buffer to log file writes in number of pages. Use this parameter when high volumes of messages are being sent through a server.
  - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **QueueManager** > *QM_name* > **Log** > **LogBufferPages**. For embedded JMS, edit the `createmq.properties` setting LogFilePages in the *install_root*\properties directory to the desired value. Run the **deletemq** and **createmq** commands to delete or recreate the queue manager. For external JMS, set LogFilePages to the desired value. Do this before creating the queue manager in the registry. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
  - **Default value:** 0
  - **Recommended value:** There is no performance penalty for setting this value too high. Therefore, set the value to its maximum of 512 pages.
– **Log primary files**
  - **Description:** Controls the number of primary or permanent log files for the queue manager. Use this parameter when high volumes of messages are being sent through a server.
  - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **QueueManager** > *QM_name* > **Log** > **LogPrimaryFiles**. For embedded JMS, edit the `createmq.properties` setting LogFilePages in the *install_root*\properties directory to the desired value. Run the **deletemq** and **createmq** commands to delete or recreate the queue manager. For external JMS, set LogPrimaryFiles to the desired value. Do this before creating the queue manager in the registry. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
  - **Default value:** External: 3, embedded: 0
  - **Recommended value:** There is no performance penalty for setting this value too high. Set the value to its maximum of 63 pages.
– **Log secondary files**
  - **Description:** Controls the number of secondary log files for the queue manager. Secondary files are files created when the primary files are not enough and deleted when they are no longer needed.
  - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **QueueManager** > *QM_name* > **Log** > **LogSecondaryFiles**. For embedded JMS, edit the `createmq.properties` setting LogSecondaryFiles in the *install_root*\properties directory to the desired value. Run the **deletemq** and **createmq** commands to delete or recreate the queue manager. For external JMS, set LogPrimaryFiles to the desired value. Do this before creating the queue manager in the registry. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
  - **Default value:** External: 2, embedded: 60
  - **Recommended value:** There is a limit of 63 total files from the primary and secondary combined and because secondary logs are slower, setting this 0 is ideal for performance.
– **Log default path**
  - **Description:** Controls the location of the queue manager log files. Use this parameter when high volumes of messages are sent through a server.

- **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVision** > **Configuration** > **QueueManager** > *QM_name* > **Log** > **LogDefaultPath**. For embedded JMS, edit the `createmq.properties` setting LogPath in the *install_root*\properties directory to the desired value. Run the **deletemq** and **createmq** commands to delete or recreate the queue manager. For external JMS, set LogDefaultPath to the desired value. Do this before creating the queue manager in the registry. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
  - **Default value:** *WebSphereMQHome*\log
  - **Recommended value:** It is ideal to have a disk dedicated to this task because WebSphere MQ tries to keep the head of the disk positioned at the place in the file where it needs to write next. A fast RAID volume is best.
– **Default queue buffer size**
  - **Description:** Controls the size in bytes of an in-memory buffer for nonpersistent queues. Use this parameter when large message sizes are used, or large bursts of messages cause the queue to back up. If the queue backs up past this buffer, messages are flushed out to the disk.
  - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVersion** > **Configuration** > **QueueManager** > *QM_name* > **TuningParameters** > **DefaultQBufferSize**. Embedded JMS is not currently supported; you need MQ5.3 CSD2 and an accompanying Version 5.0 fix. For external JMS, set DefaultQBufferSize to the desired value. Do this before creating the queue manager. Run the **amqmdain regsec** command to secure the registry value and create the queue manager.
  - **Default value:** 64K (registry key does not exist)
  - **Recommended value:** Set this parameter to accommodate the typical number of messages sitting on the queue at any given time. This should be numberOfMessages*(500+messageSizeInBytes). The maximum value is 100MB, but typically 1MB is enough.
– **Default persistent queue buffer size**
  - **Description:** Controls the size in bytes of an in-memory buffer for nonpersistent queues. Use this parameter whenever memory is available.
  - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVersion** > **Configuration** > **QueueManager** > *QM_name* > **TuningParameters** > **DefaultPQBufferSize**. Embedded JMS is not currently supported; you need MQ5.3 CSD2 and an accompanying Version 5.0 fix. For external JMS, set DefaultPQBufferSize to the desired value. Create the queue manager. This is a permanent queue setting. To make a change, delete and recreate the queue.
  - **Default value:** 0 (registry key does not exist)
  - **Recommended value:** Set this parameter to accommodate the number of typical concurrently processed messages, plus a little more for read-ahead capabilities. Do this by calculating numOfCocurrentMessages*(msgSizeInBytes+500)*2. Typically, 1MB is enough.
– **Maximum channels**
  - **Description:** Controls the allowable number of concurrent CLIENT transport clients. Use this parameter when large numbers of clients are being used.

- **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVersion** > **Configuration** > **QueueManager** > *QM_name* > **Channels** > **MaxChannels**. Embedded JMS is not currently supported; you need MQ5.3 CSD2 and an accompanying Version 5.0 fix. For external JMS, set MaxChannels to the desired value. Restart the queue manager.
  - **Default value:** External: 100 (registry key does not exist), embedded: 1000
  - **Recommended value:** Set this parameter high enough to contain the maximum number of concurrent JMS clients.
  - **Channel application bind type**
    - **Description:** Controls if the channel application is an MQ FASTPATH application. Use this parameter at all times.
    - **How to view or set:** In the registry, navigate to **HKEY_LOCAL_MACHINE** > **SOFTWARE** > **IBM** > **MQSeries** > **CurrentVersion** > **Configuration** > **QueueManager** > *QM_name* > **Channels** > **MQIBindType**. Embedded JMS is not currently supported; you need MQ5.3 CSD2 and an accompanying Version 5.0 fix. For external JMS, set MQIBindType to the desired value. Restart the queue manager.
    - **Default value:** Not FASTPATH (registry key does not exist)
    - **Recommended value:** FASTPATH
- **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. Open the administrative console.
      2. Click **Servers** > **Application Servers** > *server_name*.
      3. Click the *Server Component* > **JMS servers**.
      4. Click **Apply** or **OK**.
      5. Click **Save**.
      6. Stop and restart the application server.
    - **Default value:** 1
    - **Recommended value:** Set this value a little higher than the number of concurrent message publishers. If large numbers of subscribers exist, increasing this value can also provide some benefit.

Security

Enabling security decreases performance. The following tuning parameters give you considerations for increasing performance.
- Disable security in Global security settings
- Fine tune Cache time-out in Global security settings
- Security cache properties
- Tracking session timeout with SSL

You can also read about Sockets Layer performance tips and Tuning security configurations.

Object Request Broker

An Object Request Broker (ORB) manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It supports client requests and responses received from servers in a network-distributed environment. You can tune the ORB with the following parameters:

- Pass by reference (com.ibm.CORBA.iiop.noLocalCopies)
- Connection cache minimum (com.ibm.CORBA.MaxOpenConnections)
- Thread pool Maximum size
- com.ibm.CORBA.ServerSocketQueueDepth
- com.ibm.CORBA.FragmentSize

The Object Request Broker tuning guidelines offer tips on using these parameters to tune the ORB.

XML parser definitions

- **Description:** Facilitates server startup by adding XML parser definitions to the `jaxp.properties` and `xerces.properties` files in the ${*install_root*}/jre/lib directory. The XMLParserConfiguration value might change as new versions of Xerces are provided.
- **How to view or set:** Insert the following lines in both files:

```
javax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParserFactoryImpl
javax.xml.parsers.DocumentBuildFactory=org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
org.apache.xerces.xni.parser.XMLParserConfiguration=org.apache.xerces.parsers.
StandardParserConfiguration
```

  Note: The third and fourth lines in the preceding example should be all on one line. They are split here so that the contents fit on the page.
- **Default value:** None
- **Recommended value:** None

Dynamic cache service

Using the dynamic cache service can improve performance. See Improving performance through the dynamic cache service for information about using the dynamic cache service and how it can affect your application server performance.

Web container

One of the parts of each WebSphere Application Server is a Web container. To route servlet requests from the Web server to the Web containers, the product establishes a transport queue between the Web server plug-in and each Web container. The Web container is initially created with default property values suitable for simple Web applications. However, these values might not be appropriate for more complex Web applications. Using the following parameters, you can tune the Web container to fit the specific needs of your application server.

- Thread pool Maximum size
- Growable thread pool
- HTTP transports MaxKeepAliveConnections
- HTTP transports MaxKeepAliveRequests

EJB container

One of the parts of each application server in WebSphere Application Server is an EJB container. An EJB container is automatically created when you create an application server. After the EJB container is deployed, you can use the following parameters to make adjustments that improve performance.

- Cleanup interval
- Cache size
- Break CMP enterprise beans into several enterprise bean modules during assembly

See also EJB queue tips.

Session management

The installed default settings for session management are optimal for performance. See Tuning session management and Tuning parameter settings for more information about tuning session management.

Data sources

A data source is used to access data from the database. The following parameters reveal how the number of physical connections within a connection pool can change performance.

- **Connection pooling**
  - Maximum connection pool
  - Minimum connection pool
- Statement cache size

Hardware capacity and settings

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

- **Disk speed**
  - **Description:** Disk speed and configuration can have a dramatic effect on the performance of application servers that run applications that are heavily dependent on database support, that use extensive messaging, or are processing workflow. Using disk I/O subsystems that are optimized for performance, for example RAID array, are essential components for optimum application server performance in these environments.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value:** Spread the disk processing across as many disks as possible to avoid contention issues that typically occur with 1 or 2 disk systems.
- **Processor speed**
  - **Description:** In the absence of other bottlenecks, increasing the processor speed often helps throughput and response times.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value:** None
- **System memory**

- **Description:** Increase memory to prevent the system from paging memory to disk, improving performance. Allow a minimum of 256MB of memory for each processor. Adjust the available memory when the system is paging and processor utilization is low because of the paging.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value**: 512MB per processor
- **Networks**
  - **Description:** 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.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value:** Make sure that 100MB is in use on 10/100 Ethernet networks.

Operating systems

The following tuning parameters are specific to operating systems. Because these are not WebSphere Application Server products, be aware that the products can change and results vary.

- **Windows NT or Windows 2000TcpTimedWaitDelay**
  - **Description:** Determines the time that must elapse before TCP/IP can release a closed connection and reuse its resources. This interval between closure and release is known as the TIME_WAIT state or twice the maximum segment lifetime (2MSL) state. During this time, reopening the connection to the client and server cost less than establishing a new connection. Reducing the value of this entry allows TCP/IP to release closed connections faster, providing more resources for new connections. Adjust this parameter if the running application requires rapid release, creation of new connections, and there is a low throughput due to many connections sitting in TIME_WAIT.
  - **How to view or set:**
    1. Using the **regedit** command, access HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\TCPIP\Parameters and create a new REG_DWORD named TcpTimedWaitDelay.
    2. Set the value to decimal 30, which is Hex 0x0000001e.
    3. Stop and restart the system.
  - **Default value:** 0xF0 (240 seconds = 4 minutes)
  - **Recommended value:** The minimum value of 0x1E (30 seconds)
- **Windows NT or Windows 2000 MaxUserPort**
  - **Description:** Determines the highest port number TCP/IP can assign when an application requests an available user port from the system.
  - **How to view or set:**
    1. Using the **regedit** command, access HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\TCPIP\Parameters and create a new REG_DWORD named MaxUserPort.
    2. Stop and restart the system.
  - **Default value:** None

    – **Recommended value:** At least decimal 32768.

    Note: Use these two parameters together when tuning WebSphere Application Server on a Windows NT or Windows 2000 operating system.

- **AIX with DB2**
  - **Description:** Separating your DB2 log files from the physical database files can boost performance. You can also separate the logging and the database files from the drive containing the Journaled File System (JFS) service. AIX uses specific volume groups and file systems for the JFS logging.
  - **How to view or set:** Use the AIX `filemon` utility to view all file system input and output, and to strategically select the file system for the DB2 logs. Then, set the DB2 log location according to (logging).
  - **Default value:** The default location for the files is \home\<*db2_user_name*>\sqllib\db2dump.
  - **Recommended value:** Move the files to a disk that is separate from the DB2 data and that has minimum I/O activity.
- **AIX file descriptors (ulimit)**
  - **Description:** Specifies the number of open files permitted. The default setting is typically sufficient for most applications. If the value set for this parameter is too low, a `Memory allocation` error is displayed.
  - **How to view or set:** Check the UNIX reference pages on ulimit for the syntax of different shells. To set the ulimit to 2000 for the KornShell shell (ksh), issue the **ulimit -n 2000** command. Use the command **ulimit -a** to display the current values for all limitations on system resources.
  - **Default value:** For AIX systems, the default setting is 2000.
  - **Recommended value:** 2000
- **Other AIX information**

  There are many other AIX operating system settings to consider that are not within the scope of this document. Additional settings you can adjust follow:
  - Adapter transmit and receive queue
  - TCP/IP socket buffer
  - IP protocol mbuf pool performance
  - Update file descriptors
  - Update the scheduler

  For more information about AIX, see (for Learning).
- **Solaris file descriptors (ulimit)**
  - **Description:** Specifies the number of open files permitted. If the value of this parameter is too low, a `Too many files open` error displays in the WebSphere Application Server `stderr.log`.
  - **How to view or set:** Check the UNIX reference pages on ulimit for the syntax of different shells. For KornShell (ksh) use the **ulimit -n 1024** command. Use the **ulimit -a** command to display the current values for all limitations on system resources.
  - **Default value:** None
  - **Recommended value:** 2000
- **Solaris TCP_TIME_WAIT_INTERVAL**
  - **Description:** Notifies the TCP/IP how long to keep closed connection control blocks. After the applications complete the TCP/IP connection, the control blocks are kept for the specified time. When high connection rates occur, a large backlog of the TCP/IP connections accumulate and can slow server

performance. The server can stall during certain peak periods. If the server stalls, the **netstat** command shows that many of the sockets opened to the HTTP server are in the CLOSE_WAIT or FIN_WAIT_2 state. Visible delays can occur for up to four minutes, during which time the server does not send any responses, but CPU utilization stays high, with all of the activity in system processes.

- **How to view or set:** Use the **get** command to determine the current interval and the **set** command to specify an interval of 60 seconds. For example:

  ```
  ndd -get /dev/tcp tcp_time_wait_interval
  ndd -set /dev/tcp tcp_time_wait_interval 60000
  ```

- **Default value:** The Solaris default time wait interval is 2400000 milliseconds.

- **Recommended value:** 60000 milliseconds.

- **Solaris TCP_FIN_WAIT_2_FLUSH_INTERVAL**

  - **Description:** Specifies the timer interval prohibiting a connection in FIN_WAIT_2 to remain in that state. When high connection rates occur, a large backlog of TCP/IP connections accumulate and can slow server performance. The server can stall during peak periods. If the server stalls, using the **netstat** command shows that many of the sockets opened to the HTTP server are in CLOSE_WAIT or FIN_WAIT_2 state. Visible delays can occur for up to four minutes, during which time the server does not send any responses, but CPU utilization stays high, with all of the activity in system processes.

  - **How to view and set:** You can set the current interval to 67.5 seconds by using the following commands:

    ```
    ndd -get /dev/tcp tcp_fin_wait_2_flush_interval
    ndd -set /dev/tcp tcp_fin_wait_2_flush_interval 67500
    ```

  - **Default value:** The Solaris default is 675000.

  - **Recommended value:** 67500

- **Solaris TCP_KEEPALIVE_INTERVAL**

  - **Description:** Specifies the timer interval prohibiting an active connection from staying in ESTABLISHED state if one of the peers never responds.

  - **How to view or set:** Use the following commands to determine the current value or to set the value to 300 seconds:

    ndd -get /dev/tcp tcp_keepalive_interval ndd -set /dev/tcp tcp_keepalive_interval 300000

  - **Default value:** 7200000

  - **Recommended value:** 300000

- **Solaris kernel semsys:seminfo_semume**

  - **Description:** Limits the Max Semaphore undo entries per process. Because this setting specifies a maximum value, the parameter does not cause use of additional memory unless it is needed.

  - **How to view or set:** This value is displayed as SEMUME if the **/usr/sbin/sysdef** command is run. There can be an entry in the /etc/system file for this tuning parameter. Set this parameter through the /etc/system entry as follows:

    ```
    set semsys:seminfo_semume = 1024
    ```

  - **Default value:** 10

  - **Recommended value:** None

- **Solaris kernel semsys:seminfo_semopm**

  - **Description:** Displays as SEMOPM if the **/usr/sbin/sysdef** command is run. An entry in the /etc/system file can exist for this tuning parameter.

- **How to view or set:** Set this parameter through the /etc/system entry as follows:

  `semsys:seminfo_semopm = 200`

- **Default value:** None
- **Recommended value:** 16384

- **Java Virtual machine virtual page size**
  - **Description:** Provides the current operating system characteristics of the process executable when you set the virtual page size.
  - **How to view or set:** Set the virtual page size with the command **chatr +pi64M +pd64M /opt/WebSphere/AppServer/java/bin/PA_RISC2.0/native_threads/java**. Where *64M* stands for 64MB.
  - **Default value:** None
  - **Recommended value:** 64MB

- **HP-UX 11i**

  HP-UX 11i settings can be modified to significantly improve WebSphere Application Server performance. For additional information about the HP performance tuning parameters see (learning).

  - **Java virtual machine virtual page size**
    - **Description:** Setting the Java virtual machine page size to 64KB improves performance.
    - **How to view or set:** Use the **chatr +pi64M +pd64M /opt/WebSphere/AppServer/java/bin/PA_RISC2.0/native_threads/java** command. The command output provides the current operating system characteristics of the process executable.
    - **Default value:** 4KB, if not assigned
    - **Recommended value:** 64KB

  - **HP-UX 11i TCP_CONN_REQUEST_MAX**
    - **Description:** Specifies the maximum number of connection requests that the operating system can queue when the server does not have any available threads. When high connection rates occur, a large backlog of TCP/IP connection requests build up and client connections are dropped. Adjust this setting when clients start to timeout after waiting to connect. This situation can be verified by issuing the **netstat -p tcp** command. Look for the following value: *connect requests dropped due to full queue*
    - **How to view or set:** Set this parameter by Using the **ndd -set /dev/tcp tcp_conn_request_max 1024** command.
    - **Default value:** 20
    - **Recommended value:** 1024

  - **HP-UX 11i kernel parameter recommendations**
    - **Description:** Use the following kernel parameter settings with DB2 or ORACLE for the best performance:

Kernel parameter	WebSphere Application Server setting	DB2 setting	Oracle setting
maxuprc	—	512	—
maxfiles	2,048	—	—
maxfiles_lim	2,048	—	—
nkthreads	10,000	—	—
max_thread_proc	2,048	—	—

nproc	—	1,028	—
nflocks	—	8,192	—
ninode	—	2,048	—
nfile	—	8,192	—
msgseg	—	32,767	—
msgmnb	—	65,535	—
msgmax	—	65,535	—
msgtql	—	1,024	—
msgmap	—	258	—
msgmni	—	256	—
msgssz	—	16	—
semmni	—	512	70
semmap	—	514	—
semmns	—	1,024	200
semmnu	—	1,024	—
shmmax	—	966,367,642	1 GB
shmmseg	—	16	10
shmmni	—	300	100

- **How to view or set:** Use the HP-UX SAM utility to set the kernel parameters. See the documentation for your operating system for directions.
- **Default value:** None
- **Recommended value:** See table
– **HP-UX 11i kernel parameter recommendations for WebSphere MQ 5.3**
- **Description:** The embedded messaging uses WebSphere MQ 5.3. The following are WebSphere MQ 5.3 recommended kernel parameter settings:

Kernel parameter	Setting
shmmax	536870912
shmseg	1024
shmmni	1024
shmem	1
sema	1
semaem	16384
semvmx	32767
semmns	16384
semmni	1024 (semmni < semmns)
semmap	1026 (semmni +2)
semmnu	2048
semume	256
msgmni	50
msgtql	256
msgmap	258 (msgtql +2)

msgmax	4096
msgmnb	4096
msgssz	8
msgseg	1024
maxusers	32
max_thread_proc	66
maxfiles	1024
nfile	10000

- **How to view or set:** Use the HP-UX SAM utility to set the kernel parameters. See the documentation for your operating system for directions.
- **Default value:** None
- **Recommended value:** See table
- **Generational garbage collection nursery size**
  - **Description:** WebSphere Application Server 5.0 ships with the the HP native JVM, which is based on the Sun Hotspot technology. One of its features is the use of generational garbage collection where the heap is divided into new and old generations. The appropriate size of the new generation or nursery must be determined using performance analysis tools such as Glance. If the nursery size is properly chosen, the overhead of garbage collection is reduced and throughput and response time improve.
  - **How to view or set:** Use the **-Xmn** command at the generic Java options, for example, using **-Xmn512m** sets the nursery size to 512MB.
  - **Default value:** One-third the maximum heap size.
  - **Recommneded value:** When several short-life objects are created, set the value to one-half the maximum heap size.

Web servers

(Web server tuning parameters) lists tuning parameters specific to Web servers. Because these are not WebSphere Application Server products be aware that the products can change and results vary.

Performance troubleshooting tips

Review the symptom table for information to help you troubleshoot performance problems.

Problem	Tuning parameters
Overall throughput and response time are undesirable.	Processor speed
AIX: Memory allocation error	AIX file descriptors (ulimit) Scroll down to this parameter from Operating Systems.
ORB: Response time and throughput indicate EJB requests with shorter execution times are being denied adequate access to threads in ORB thread pool.	Object Request Broker service settings that can be added to the administrative console, Logical Pool Distribution (LPD) mechanism, set using ORB custom properties (with link to rejb_setg.xml, keyref = rorbsetg)

Linux: Default maximum web container threads set too high. For Linux distributions that are based on the 2.4 and earlier kernels, setting the web container maximum thread size too high will degrade performance. The maximum web container thread should be set to 25. A higher web container thread size will cause too much contention, therefore degrading performance.	
Solaris operating environment: Too many files open	Solaris file descriptors (ulimit) Scroll down to this parameter from Operating Systems.
Solaris operating environment: The server stalls during peak periods, responses take minutes, processor utilization remains high with all activity in the system processes, and netstat shows many sockets are open to port 80 in CLOSE_WAIT or FIN_WAIT_2 state.	Solaris tcp_time_wait_interval and Solaris tcp_fin_wait_2_flush_interval Scroll down to this parameter from Operating Systems.
Windows NT or Windows 2000: Netstat shows too many sockets are in TIME_WAIT.	Windows NT or Windows 2000 TcpTimedWaitDelay.
Under load, client requests do not arrive at the Web server because they time out or are rejected.	For IBM HTTP Server on Windows NT, see ListenBackLog.
Windows NT or Windows 2000: WebSphere Application Server performance decreased after an application server from another vendor was installed.	Microsoft Internet Information Server (IIS) properties.
The Percent Maxed metric from the Tivoli Performance Viewer indicates that the Web container thread pool is too large or too small.	Thread pool Maximum size.
Netstat shows too many TIME_WAIT state sockets for port 9080.	HTTP transports MaxKeepAliveConnections and HTTP transports MaxKeepAliveRequests.
Too much disk input and output occurs due to paging.	Maximum heap size.
The Percent Used metric for a data source connection pool from the Tivoli Performance Viewer indicates the pool size is too large.	Maximum connection pool and Minimum connection pool.
The Prepared Statement Discards metric from the Tivoli Performance Viewer indicates that the data source statement cache size is too small.	Statement cache size.
Too much disk input and output occurs due to DB2 writing log records.	DB2 MinCommit.
The Percent Maxed metric from the Tivoli Performance Viewer indicates that the Object Request Broker thread pool is too small.	EJB method Invocation Queuing.
The Java Virtual Machine Profiler Interface (JVMPI) from the Tivoli Performance Viewer indicates over-utilization of objects when too much time is spent in garbage collection.	Detecting over-utilization of objects.

The Used Memory metric from the Tivoli Performance Viewer shows memory leaks and the Java code displays an `Out of Memory` exception.	Detecting memory leaks.
Throughput, response time and scalability are undesirable.	If the application permits, exploit the dynamic cache service.
Startup performance is poor.	Using Java virtual machine settings, set a minimum heap size of at least 50MB (-Xms50m).

Using the Runtime Performance Advisor

5.0.2 The Runtime Performance Advisor provides advice to help tune systems for optimal performance and is configured using the WebSphere Application Server administrative console. The Runtime Performance Advisor uses Performance Monitoring Infrastructure (PMI) data to provide recommendations for performance tuning. Running in the JVM of the application server, this advisor periodically checks for inefficient settings, and issues recommendations as standard product warning messages. These recommendations are displayed both as warnings in the administrative console under WebSphere Runtime Messages in the WebSphere Status panel and as text in the application server `SystemOut.log` file. Enabling the Runtime Performance Advisor has minimal system performance impact.

Steps for this task

1. Enable PMI services in WebSphere Application Server through the administrative console, and Enable PMI services in NodeAgent through the administrative console if running WebSphere Application Server Network Deployment.

   In order to obtain advice, you must first enable the performance monitoring service through the administrative console and restart the server. If running Network Deployment, you *must* enable PMI service on both the server and on the node agent and restart the server and node agent. The Runtime Performance Advisor enables the appropriate monitoring counter levels for all enabled advice. If there are specific counters that are not wanted, disable the corresponding advice in the Runtime Performance Advisor Panel, and disable unwanted counters.

2. Enable PMI services in NodeAgent through the administrative console. In order to obtain advice, you *must* enable PMI service on both the server and on the node agent, and restart the server and node agent.

3. Start the administrative console.

4. Click **Servers** > **Application Servers** in the console navigation tree.

5. Click *server_name* > **Runtime Performance Advisor Configuration**.

6. Click the **Configuration** tab.

7. Select the **Number of Processors**.

   Select the appropriate settings for your system configuration to ensure accurate advice.

8. **(Optional)** Select the **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.

9. **(Optional)** 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 only issued if the rate of discards exceeds the new threshold setting.

10. Click **Apply**.

11. Click **Save**.

12. Click the **Runtime** tab.

13. Click **Restart**.

    Selecting **Restart** on the Runtime tab reinitializes the Runtime Performance Advisor using the last configuration information saved to disk.

    Note: This action also resets the state of the Runtime Performance Advisor. For example, the current warning count is reset to zero for each message.

14. 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. The Runtime Performance Advisor only provides advice when CPU utilization exceeds a sufficiently high level. For a list of IBM business partners providing tools to drive this type of load, see the article, Performance: Resources for learning in the sub-section of Monitoring performance with third party tools.

15. 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.

    Note: Enable the Runtime Performance Advisor once conditions have reached a fully loaded condition to achieve the best results for performance tuning.

16. Click **OK**.

17. Select **Warnings** in the administrative console under the WebSphere Runtime Messages in the WebSphere Status panel or look in the SystemOut.log file, located in the *install_root*\logs\servername directory to view tuning advice. Some messages are not issued immediately.

18. Update the product configuration for improved performance, based on advice.

    Although the performance advisors attempt to distinguish between loaded and idle conditions, misleading advice might be issued if the advisor is enabled while the system is ramping up or down. 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 ensure it functions and performs well.

What to do next

WebSphere Application Server also allows you to enable and disable advice in the Advice Configuration panel. 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 on Advice configuration, see the article, Advice configuration
settings.

Runtime Performance Advisor configuration settings

5.0.2 Use this page to specify settings for the Runtime Performance Advisor.

To view this administrative page, click **Servers** > **Application Servers** >
server_name > **Runtime Performance Advisor Configuration**.

Enable Runtime Performance Advisor
Specifies whether the Runtime Performance Advisor runs.

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 will appear as averages over this
interval.

Maximum warning sequence
Specifies the number of consecutive warnings issued before the threshold is
updated.

Number of processors
Specifies the number of processors on the server.

Restart button
Selecting Restart on the Runtime tab reinitializes the Runtime Performance Advisor
using the last information saved to disk. Note that this action also resets the state
of the Runtime Performance Advisor. For example, the current warning count is
reset to zero for each message.

Advice configuration settings

5.0.2 Use this page to select the advice you wish to enable or disable.

To view this administrative page, click **Servers** > **Application Servers** >
server_name > **Runtime Performance Advisor Configuration** > **Advice
Configuration**.

Advice name
Specifies the advice that you can enable or disable.

Advice applied to component
Specifies the WebSphere Application Server component to which the runtime
performance advice applies.

Advice status
Specifies whether advice is stopped or started.

There are only two values — **Started** and **Stopped**. **Started** means that the advice
runs if the advice applies. **Stopped** means that the advice does not run.

Advice status
Specifies whether advice is stopped, started or unavailable.

The advice status has one of three values — **Started**, **Stopped** or **Unavailable**. **Started** means that the advice is being applied. **Stopped** means that the advice is not applied. **Unavailable** means that the advice does not apply to the current configuration (such as Persisted Session Size advice in a configuration without persistent sessions).

Using the Performance Advisor in Tivoli Performance Viewer

5.0.2 The Performance Advisor in Tivoli Performance Viewer (TPV) provides advice to help tune systems for optimal performance and gives recommendations on inefficient settings by using collected Performance Monitoring Infrastructure (PMI) data. Advice is obtained by selecting the Performance Advisor icon in TPV. The Performance Advisor in TPV provides more extensive advice than the Runtime Performance Advisor. For example, TPV provides advice on setting the dynamic cache size, setting the JVM heap size and using the DB2 Performance Configuration Wizard.

Steps for this task

1. Enable PMI services in WebSphere Application Server through the administrative console, and Enable PMI services in NodeAgent through the administrative console

   In order to obtain advice, you must first enable the performance monitoring service through the administrative console and restart the server. If running Network Deployment, you *must* enable PMI service on both the server and on the node agent and restart the server and node agent.

2. Enable data collection.

   The monitoring levels that determine which data counters are enabled can be set dynamically, without restarting the server. These monitoring levels and the data selected determine the type of advice you obtain. The Performance Advisor in TPV uses the standard monitoring level; however, the Performance Advisor in TPV can use a few of the more expensive counters (to provide additional advice) and provide advice on which counters can be enabled. This action can be completed in one of the following ways:

   a. Enable data collection through the administrative console.

   b. Enable performance monitoring services through Tivoli Performance Viewer.

   c. Enable performance monitoring services using the command line.

3. Start the Tivoli Performance Viewer.

4. Simulate a production level load.

   Simulate a realistic production load for your application, if you are using the Performance Advisor in a test environment, or doing any other performance tuning. 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. The Performance Advisor only provides advice when CPU utilization exceeds a sufficiently high level. For a list of IBM business partners providing tools to drive this type of load, see the article, Performance: Resources for learning in the sub-section of Monitoring performance with third party tools.

5. **(Optional)** Store data to a log file.

6. **(Optional)** Replay a performance data log file.

7. **(Optional)** Refresh data.

Clicking refresh with `server` selected under the **viewer** icon causes TPV to:

- Query the server for new PMI and product configuration information.

Click refresh with `server` selected under the **advisor** icon causes TPV to:

- Refresh advice that is provided in a single instant in time.
- Not query the server for new PMI and product configuration information.

8. Tuning advice appears when the Advisor icon is chosen in the TPV Performance Advisor. Double-click an individual message for details.

   Since PMI data is taken over an interval of time and averaged to provide advice, details within the advice message appear as averages.

9. Update the product configuration for improved performance, based on advice. Since Tivoli Performance Viewer refreshes advice at a single instant in time, take the advice from the peak load time.

   Although the performance advisors attempt to distinguish between loaded and idle conditions, misleading advice might be issued if the advisor is enabled while the system is ramping up or down. 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 ensure it functions and performs well.

10. **(Optional)** Clear values from tables and charts.
11. **(Optional)** Reset counters to zero.

Performance Advisor Report in Tivoli Performance Viewer

5.0.2 View recommendations and data from the Performance Advisor in Tivoli Performance Viewer by expanding the Performance Advisor icon under Data Collection in Tivoli Performance Viewer and selecting the server.

Message
Specifies recommendations for performance tuning.

Double click the message to obtain more details.

Performance data
Displays performance data for the WebSphere Application Server.

The first table represents the number of requests per second and the response time in milliseconds for both the Web and EJB containers.

The pie graph displays the processor activity, percentage busy and idle.

The bar graph displays the total size and number of busy and idle threads and connections in the Web container thread pool, the Object Request Broker (ORB) Service thread pool and any database connection pools.