WebSphere® Business Integration Server Foundation, Version 5.1

Performance Monitoring and Tuning

Note
Before using this information, be sure to read the general information under "Notices" on page 107.

Compilation date: April 19, 2004

Contents

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback** .
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-0206.

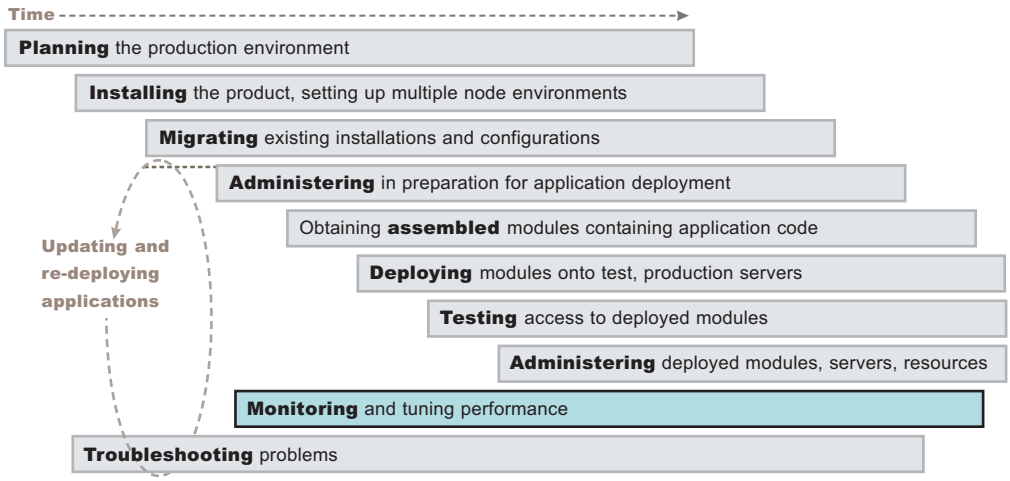
  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Chapter 1. Welcome to Monitoring

Performance monitoring is an activity in which you collect and analyze data about the performance of your applications and their environment.

A simple time line of activities for Planning, Installer and Administrator roles.



To help administrators identify application performance problems, the product collects performance data and supplies interfaces that allow external applications to monitor the performance data; it also provides tools to display performance data for analysis.

The product collects data on run-time and applications through the Performance Monitoring Infrastructure (PMI), as described in "Performance Monitoring Infrastructure" on page 3. This infrastructure is compatible with and extends the JSR-077 specification.

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)" on page 34.

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 Monitoring (ARM) agents used by Tivoli or third party monitoring tools.

For more information about PMI Request Metrics, see “Measuring data requests (Performance Monitoring Infrastructure Request Metrics)” on page 82.

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.

To set up performance monitoring:

1. "Enabling performance monitoring services in the application server through the administrative console" on page 29 and "Enabling performance monitoring services in the Node Agent through the administrative console" on page 30 if running WebSphere Application Server Network Deployment. To monitor performance data through the PMI interfaces, you must first enable the performance monitoring service through the administrative console before restarting the server. If running in Network Deployment, you need to enable PMI services on both the server and on the node agent before restarting the server and the node agent.
2. Collect the data.

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

   a. "Starting the Tivoli Performance Viewer" on page 39.
   b. "Enabling performance monitoring services using the command line" on page 30.

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)" on page 82.

Related tasks

"Performance monitoring service settings" on page 29

"Starting the Tivoli Performance Viewer" on page 39

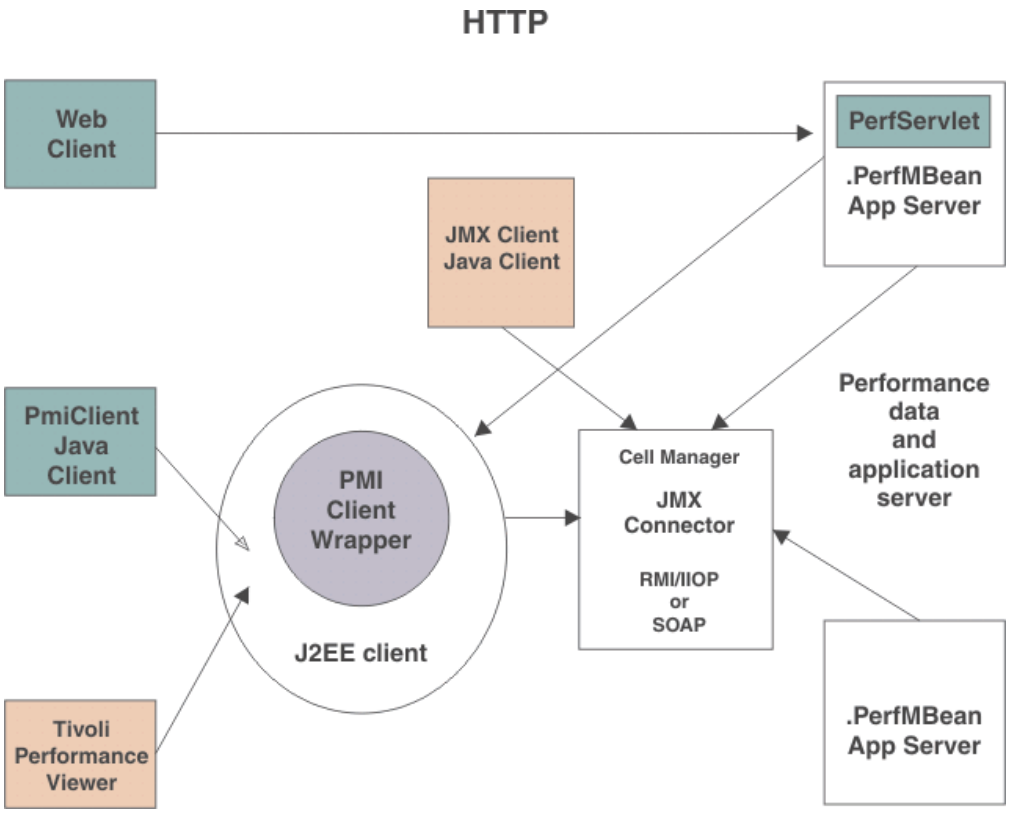
Using the dynamic cache service to improve performance

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. Refer to Performance Monitoring Infrastructure client and Performance Monitoring Infrastructure interface.

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, a Java client, or a Java Management Extensions (JMX) client. WebSphere Application Server contains Tivoli Performance Viewer, a Java client which displays and monitors performance data. See the 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 topics for more information on monitoring tools.

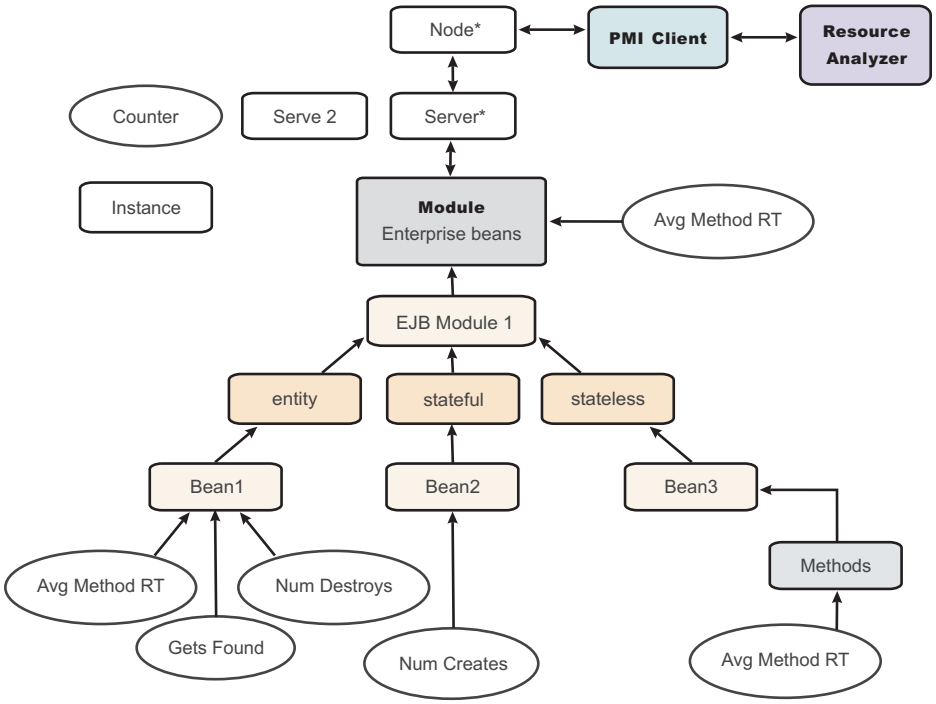


The figure shows the overall PMI architecture. On the right side, the server updates and keeps PMI data in memory. The left side displays a Web client, a Java client, and a JMX client retrieving the performance data.

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.

Hierarchy of data collections used for performance reporting to Resource Analyzer



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.

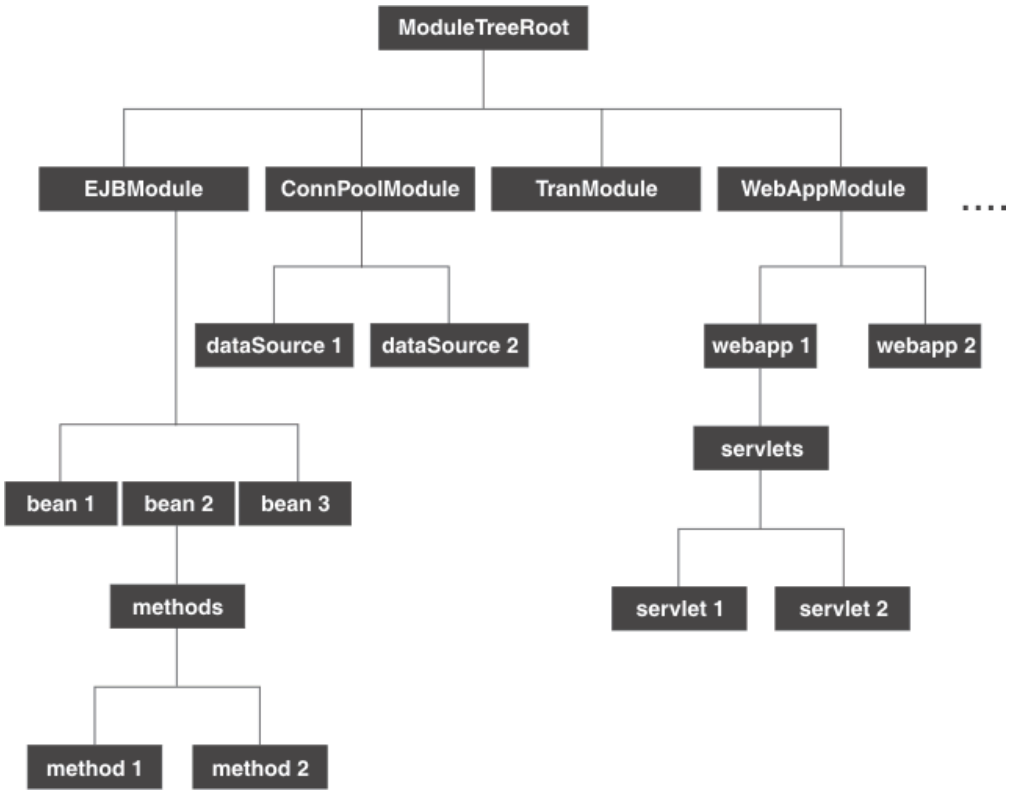
The Tivoli Performance Viewer allows users to view and manipulate the data for counters. A particular counter type can appear in several modules. For example, both the servlet and enterprise bean modules have a response time counter. In

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 no longer includes 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 is not 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.

Java 2 Connector (J2C) connection pool
Data counters for this category contain usage information about the Java 2 Platform 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, the number of invalidations, and the number of hits and misses.

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

Alarm manager
Data counters for this category contain information for the Alarm Manager.

Object pool
Data counters for this category contain information for Object Pools.

Scheduler
Data counters for this category contain information for the Scheduler service.

You can access PMI data through the getStatsObject and the getStatsArray method in the PerfMBean. You need to pass the MBean ObjectName(s) to the PerfMBean.

Use the following MBean types to get PMI data in the related categories:

- DynaCache: dynamic cache PMI data
- EJBModule*: Enterprise Java Bean (EJB) module PMI data (BeanModule)
- EntityBean*: specific EJB PMI data (BeanModule)
- JDBCProvider*: Java Database Connectivity (JDBC) connection pool PMI data
- J2CResourceAdapter*: Java 2 Connectivity (J2C) connection pool PMI data
- JVM: Java virtual machine PMI data
- MessageDrivenBean*: specific EJB PMI data (BeanModule)
- ORB: Object Request Broker PMI data
- Server: PMI data in the whole server, you must pass recursive=true to PerfMBean
- SessionManager*: HTTP Sessions PMI data
- StatefulSessionBean*: specific EJB PMI data (BeanModule)
- StatelessSessionBean*: specific EJB PMI data (BeanModule)
- SystemMetrics: system level PMI data
- ThreadPool*: thread pool PMI data

- TransactionService: JTA Transaction PMI data
- WebModule*: Web application PMI data
- Servlet*: servlet PMI data
- WLMAppServer: Workload Management PMI data
- WebServicesService: Web services PMI data
- WSGW*: Web services gateway PMI data

To use the AdminClient API to query the MBean ObjectName for each MBean type. You can either query all the MBeans and then match the MBean type or use the query String for the type only: `String query = "WebSphere:type=mytype,node=mynode,server=myserver,*";`

Set the mytype, mynode, and myserver values accordingly. You get a Set value when you call the AdminClient class to query MBean ObjectNames. This response means that you can get multiple ObjectNames.

In the previous example, the MBean types with a star (*) mean that there can be multiple ObjectNames in a server for the same MBean type. In this case, the ObjectNames can be identified by both type and name (but mbeanIdentifier is the real UID for MBeans). However, the MBean names are not predefined. They are decided at run time based on the applications and resources. When you get multiple ObjectNames, you can construct an array of ObjectNames that you are interested in. Then you can pass the ObjectNames to PerfMBean to get PMI data. You have the recursive and non-recursive options. The recursive option returns Stats and sub-stats objects in a tree structure while the non-recursive option returns a Stats object for that MBean only. More programming information can be found in "Develop your own monitoring applications".

Enterprise Java Bean counters

Data counters for this category report load values, response times, and life cycle activities for enterprise beans.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Num creates	beanModule.creates	Number of times that beans were created	3.5.5 and later	Per home	CountStatistic	Low
Num removes	beanModule.removes	Number of times that beans were removed	3.5.5 and later	Per home	CountStatistic	Low
Num passivates	beanModule.passivates	Number of times that beans were passivated (entity and stateful)	3.5.5 and later	Per home	CountStatistic	Low
Num activates	beanModule.activates	Number of times that beans were activated (entity and stateful)	3.5.5 and later	Per home	CountStatistic	Low
Num loads	beanModule.loads	Number of times that bean data was loaded from persistent storage (entity)	3.5.5 and later	Per home	CountStatistic	Low
Num stores	beanModule.stores	Number of times that bean data was stored in persistent storage (entity)	3.5.5 and later	Per home	CountStatistic	Low

Num instantiates	beanModule.instantiates	Number of times that bean objects were instantiated	3.5.5 and later	Per home	CountStatistic	Low
Num destroys	beanModule.destroys	Number of times that bean objects were freed	3.5.5 and later	Per home	CountStatistic	Low
Ready beans	beanModule.readyCount	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 later	Per home	RangeStatistic	High
Concurrent lives	beanModule.concurrentLives	Number of concurrently live beans	3.5.5 and later	Per home	RangeStatistic	High
Avg method RT (ms)	beanModule.avgMethodRt	Average response time in milliseconds on the bean methods (home, remote, local)	3.5.5 and later	Per home	TimeStatistic	High
Avg create time (ms)	beanModule.avgCreateTime	Average time in milliseconds that a bean create call takes including the time for the load if any	5.0	Per home	TimeStatistic	Max
Avg load time (ms)	beanModule.loadTime	Average time in milliseconds for loading the bean data from persistent storage (entity)	5.0	Per home	TimeStatistic	Medium
Avg store time (ms)	beanModule.storeTime	Average time in milliseconds for storing the bean data to persistent storage (entity)	5.0	Per home	TimeStatistic	Medium
Avg remove time (ms)	beanModule.avgRemoveTime	Average time in milliseconds that a bean entries call takes including the time at the database, if any	5.0	Per home	TimeStatistic	Max
Total method calls	beanModule.totalMethodCalls	Total number of method calls	3.5.5 and later	Per home	CountStatistic	High
Activation time (ms)	beanModule.activationTime	Average time in milliseconds that a beanActivate call takes including the time at the database, if any	5.0	Per home	TimeStatistic	Medium
Passivation time (ms)	beanModule.passivationTime	Average time in milliseconds that a beanPassivate call takes including the time at the database, if any	5.0	Per home	TimeStatistic	Medium
Active methods	beanModule.activeMethods	Number of concurrently active methods - the number of methods called at the same time.	3.5.5 and later	Per home	TimeStatistic	High

Gets from pool	beanModule.getsFromPool	Number of calls retrieving an object from the pool (entity and stateless)	3.5.5 and later	Per home and per object pool	CountStatistic	Low
Gets found	beanModule.getsFound	Number of times that a retrieve found an object available in the pool (entity and stateless)	3.5.5 and later	Per home and per object pool	CountStatistic	Low
Returns to pool	beanModule.returnsToPool	Number of calls returning an object to the pool (entity and stateless)	3.5.5 and later	Per home and per object pool	CountStatistic	Low
Returns discarded	beanModule.returnsDiscarded	Number of times that the returning object was discarded because the pool was full (entity and stateless)	3.5.5 and later	Per home and per object pool	CountStatistic	Low
Drains from pool	beanModule.drainsFromPool	Number of times that the daemon found the pool was idle and attempted to clean it (entity and stateless)	3.5.5 and later	Per home and per object pool	CountStatistic	Low
Avg drain size	beanModule.avgDrainSize	Average number of objects discarded in each drain (entity and stateless)	3.5.5 and later	Per home and per object pool	TimeStatistic	Medium
Pool size	beanModule.poolSize	Number of objects in the pool (entity and stateless)	3.5.5 and later	Per home and per object pool	RangeStatistic	High
Message count	beanModule.messageCount	Number of messages delivered to the bean onMessage method (message driven beans)	5.0	Per type	CountStatistic	Low
Message backout count	beanModule.messageBackoutCount	Number of messages that failed to be delivered to the bean onMessage method (message driven beans)	5.0	Per type	CountStatistic	Low
Server session wait time (ms)	beanModule.avgSrvSessionWaitTime	Average time to obtain a ServerSession from the pool (message drive bean)	5.0	Per type	TimeStatistic	Medium
Server session usage	beanModule.serverSessionUsage	Percentage of the server session pool in use (message driven)	5.0	Per type	RangeStatistic	High

Enterprise Java Bean method counters

Counter definitions:

Name	Key	Description	Version	Granularity	Type	Level

Method Calls	beanModule.methods.methodCalls	Number of calls to the bean methods (home, remote, local)	3.5.5 and later	Per method or per home	CountStatistic	Max
Method Response Time (ms)	beanModule.methods.methodRt	Average response time in milliseconds on the bean methods (home, remote, local)	3.5.5 and later	Per method or per home	TimeStatistic	Max
Concurrent Invocations	beanModule.method.methodLoad	Number of concurrent invocations to call a method	5.0	Per method or per home	RangeStatistic	Max

JDBC connection pool counters

PMI collects performance data for 4.0 and 5.0 Java Database Connectivity (JDBC) data sources. For a 4.0 data source, the data source name is used. For a 5.0 data source, the Java Naming and Directory Interface (JNDI) name is used.

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

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Num creates	connectionPoolModule.numCreates	Total number of connections created	3.5.5 and later	Per connection pool	CountStatistic	Low
Pool size	connectionPoolModule.poolSize	The size of the connection pool	3.5.5 and later	Per connection pool	BoundedRangeStatistic	High
Free pool size	connectionPoolModule.freePoolSize	The number of free connections in the pool	5.0	Per connection pool	BoundedRangeStatistic	High
Num allocates	connectionPoolModule.numAllocates	Total number of connections allocated	3.5.5 and later	Per connection pool	CountStatistic	Low
Num returns	connectionPoolModule.numReturns	Total number of connections returned	4.0 and later	Per connection pool	CountStatistic	Low
Concurrent waiters	connectionPoolModule.concurrentWaiters	Number of threads that are currently waiting for a connection	3.5.5 and later	Per connection pool	RangeStatistic	High
Faults	connectionPoolModule.faults	Total number of faults, such as timeouts, in the connection pool	3.5.5 and later	Per connection pool	CountStatistic	Low
Num closes	connectionPoolModule.numDestroys	Number of times bean objects were freed	3.5.5 and later	Per connection pool	CountStatistic	Low
Avg wait time (ms)	connectionPoolModule.avgWaitTime	Average waiting time in milliseconds until a connection is granted	5.0	Per connection pool	TimeStatistic	Medium

Avg use time (ms)	connectionPoolModule.avgUseTime	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	connectionPoolModule.percentUsed	Average percent of the pool that is in use	3.5.5 and later	Per connection pool	RangeStatistic	High
Percent maxed	connectionPoolModule.percentMaxed	Average percent of the time that all connections are in use	3.5.5 and later	Per connection pool	RangeStatistic	High
Prepared stmt cache discards	connectionPoolModule.prepStmtCacheDiscards	Total number of statements discarded by the LRU algorithm of the statement cache	4.0 and later	Per connection pool	CountStatistic	Low
Num managed connections	connectionPoolModule.numManagedConnections	Number of ManagedConnection objects in use for a particular connection pool (apply to 5.0 DataSource only)	5.0	Per connection factory	CountStatistic	Low
Num connection handles	connectionPoolModule.numConnectionHandles	Number of Connection objects in use for a particular connection pool (apply to 5.0 DataSource only)	5.0	Per connection factory	CountStatistic	Low
JDBC time	connectionPoolModule.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 counters

The Java 2 Connector (J2C) connection pool counters are used to monitor the J2C connection pool performance. You can find the data using the Tivoli Performance Viewer and by looking under each application server. Click ***application_server*** > **J2C connection pool**.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Num managed connections	j2cModule.numManagedConnections	Number of ManagedConnection objects in use	5.0	Per connection factory	CountStatistic	Low
Num connection handles	j2cModule.numConnectionHandles	Number of connections that are associated with ManagedConnections (physical connections) in this pool	5.0	Per connection factory	CountStatistic	Low

Num Connections Created	j2cModule.numManagedConnectionsCreated	Total number of managed connections created	5.0	Per connection factory	CountStatistic	Low
Num Connections Destroyed	j2cModule.numManagedConnectionsDestroyed	Total number of managed connections destroyed	5.0	Per connection factory	CountStatistic	Low
Num Connections Allocated	j2cModule.numManagedConnectionsAllocated	Total number of times a managed connection is allocated to a client (the total is maintained across the pool, not per connection).	5.0	Per connection factory	CountStatistic	Low
Num Connections Freed	j2cModule.numManagedConnectionsReleased	Total number of times a managed connection is released back to the pool (the total is maintained across the pool, not per connection).	5.0	Per connection factory	CountStatistic	Low
Num Faults	j2cModule.faults	Number of faults, such as timeouts, in connection pool	5.0	Per connection factory	CountStatistic	Low
Free Pool Size	j2cModule.freePoolSize	Number of free connections in the pool	5.0	Per connection factory	BoundedRangeStatistic	High
Pool Size	j2cModule.poolSize	Average number of managed connections in the pool.	5.0	Per connection factory	BoundedRangeStatistic	High
Concurrent Waiters	j2cModule.concurrentWaiters	Average number of threads concurrently waiting for a connection	5.0	Per connection factory	RangeStatistic	High
Percent Used	j2cModule.percentUsed	Average percent of the pool that is in use	5.0	Per connection factory	RangeStatistic	High
Percent Maxed	j2cModule.percentMaxed	Average percent of the time that all connections are in use	5.0	Per connection factory	RangeStatistic	High
Avg Wait Time	j2cModule.avgWait	Average waiting time in milliseconds until a connection is granted	5.0	Per connection factory	TimeStatistic	Medium
Avg Use Time	j2cModule.useTime	Average time in milliseconds that connections are in use	5.0	Per connection factory	TimeStatistic	Medium

Java Virtual Machine counters

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

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Free memory (KB)	jvmRuntimeModule.freeMemory	Free memory in JVM run time	3.5.5 and above	Per Java Virtual Machine (JVM)	CountStatistic	Low
Used memory (KB)	jvmRuntimeModule.usedMemory	Used memory in JVM run time	3.5.5 and above	Per JVM	CountStatistic	Low

Total memory (KB)	jvmRuntimeModule.totalMemory	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
JVM up time (seconds)	jvmRuntimeModule.upTime	The amount of time the JVM is running	5.0	Per JVM	CountStatistic	Low
Num GC calls	jvmRuntimeModule.numGcCalls	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
Avg time between GC calls (ms)	jvmRuntimeModule.avgTimeBetweenGcCalls	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
Avg GC duration (ms)	jvmRuntimeModule.avgGcDuration	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 lock	jvmRuntimeModule.numWaitsForLock	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 wait for lock	jvmRuntimeModule.avgTimeWaitForLock	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
Num objects allocated	jvmRuntimeModule.numObjectsAllocated	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
Num objects moved	jvmRuntimeModule.numObjectsMoved	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
Num objects freed	jvmRuntimeModule.numObjectsFreed	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
Num started threads	jvmRuntimeModule.numThreadsStarted	Number of threads started. This counter is not available unless -XrunpmiJvmpiProfiler is set when starting the JVM.	4.0 and above	Per JVM	CountStatistic	Max

Num dead threads	jvmRuntimeModule.numThreadsDead	Number of threads dies. This counter is not available unless -XrunpmiJvmpiProfiler is set when starting the JVM.	4.0 and above	Per JVM	CountStatistic	Max

Object Request Broker counters

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Lookup time	orbPerfModule.referenceLookupTime	The time (in milliseconds) to look up an object reference before method dispatch can be carried out. Excessively long time may indicate an EJB container lookup problem.	5.0	Object Request Broker (ORB)	TimeStatistic	Medium
Total requests	orbPerfModule.totalRequests	The total number of requests sent to the ORB	5.0	ORB	CountStatistic	Low
Concurrent requests	orbPerfModule.concurrentRequests	The number of requests that are concurrently processed by the ORB	5.0	ORB	RangeStatistic	High
Processing time	orbPerfModule.interceptors.processingTime	The time (in milliseconds) it takes a registered portable interceptor to run	5.0	Per interceptor	TimeStatistic	Medium

Servlet session counters

Data counters for this category contain usage information for HTTP sessions.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Created sessions	servletSessionsModule.createdSessions	Number of sessions that were created	3.5.5 and later	Per web application	CountStatistic	Low
Invalidated sessions	servletSessionsModule.invalidatedSessions	Number of sessions that were invalidated	3.5.5 and later	Per web application	CountStatistic	Low
Session life time (ms)	servletSessionsModule.sessionLifeTime	The average session life time in milliseconds (time invalidated - time created)	3.5.5 and later	Per web application	TimeStatistic	Medium
Active sessions	servletSessionsModule.activeSessions	The number of concurrently active sessions. A session is active if the WebSphere Application Server is currently processing a request which uses that session.	3.5.5 and later	Per web application	RangeStatistic	High
Live sessions	servletSessionsModule.liveSessions	The number of sessions that are currently cached in memory	5.0 and later	Per web application	RangeStatistic	High

No room for new sessions	servletSessionsModule.noRoomForNewSession	Applies only to session in memory with AllowOverflow=false. The number of times that a request for a new session cannot be handled because it would exceed the maximum session count.	5.0	Per Web application	CountStatistic	Low
Cache discards	servletSessionsModule.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
External read time	servletSessionsModule.externalReadTime	Time (milliseconds) taken in reading the session data from the persistent store. For multirow sessions, the metrics are for the attribute; for single row sessions, the metrics are for the entire session. Applicable only for persistent sessions. When using a JMS persistent store, you choose to serialize the replicated data. If you choose not to serialize the data, the counter is not available.	5.0	Per Web application	TimeStatistic	Medium
External read size	servletSessionsModule.externalReadSize	Size of the session data read from persistent store. Applicable only for (serialized) persistent sessions; similar to external Read Time.	5.0	Per Web application	TimeStatistic	Medium
External write time	servletSessionsModule.externalWriteTime	Time (milliseconds) taken to write the session data from the persistent store. Applicable only for (serialized) persistent sessions. Similar to external Read Time.	5.0	Per Web application	TimeStatistic	Medium
External write size	servletSessionsModule.externalWriteSize	Size of the session data written to persistent store. Applicable only for (serialized) persistent sessions. Similar to external Read Time.	5.0	Per Web application	TimeStatistic	Medium
Affinity breaks	servletSessionsModule.affinityBreaks	The number of requests received for sessions that were last accessed from another Web application. This value can indicate failover processing or a corrupt plugin configuration.	5.0	Per Web application	CountStatistic	Low

Session object size	servletSessionsModule.serializableSessObjSize	The size in bytes of (the serializable attributes of) in-memory sessions. Only session objects that contain at least one serializable attribute object is counted. 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
Time since last activated	servletSessionsModule.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
Invalidated via timeout	servletSessionsModule.invalidatedViaTimeout	The number of sessions that are invalidated via timeout.	5.0	Per Web application	CountStatistic	Low
Activate non-exist sessions	servletSessionsModule.activateNonExistSessions	Number of requests for a session that no longer exists, presumably 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 counters

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Global trans begun	transactionModule.globalTransBegun	Total number of global transactions started on the server	4.0 and later	Per transaction manager or server	CountStatistic	Low
Global trans involved	transactionModule.globalTransInvolved	Total number of global transactions involved on the server (for example, begun and imported)	4.0 and later	Per transaction manager or server	CountStatistic	Low
Local trans begun	transactionModule.localTransBegun	Total number of local transactions started on the server	4.0 and later	Per transaction manager or server	CountStatistic	Low
Active global trans	transactionModule.activeGlobalTrans	Number of concurrently active global transactions	3.5.5 and later	Per transaction manager or server	CountStatistic	Low
Active local trans	transactionModule.activeLocalTrans	Number of concurrently active local transactions	4.0 and later	Per transaction manager or server	CountStatistic	Low
Global tran duration	transactionModule.globalTranDuration	Average duration of global transactions	3.5.5 and later	Per transaction manager or server	TimeStatistic	Medium
Local tran duration	transactionModule.localTranDuration	Average duration of local transactions	4.0 and later	Per transaction manager or server	TimeStatistic	Medium
Global before completion duration	transactionModule.globalBeforeCompletionDuration	Average duration of before_completion for global transactions	4.0 and later	per transaction manager or server	TimeStatistic	Medium
Global commit duration	transactionModule.globalCommitDuration	Average duration of commit for global transactions	4.0 and later	Per transaction manager or server	TimeStatistic	Medium

Global prepare duration	transactionModule.globalPrepareDuration	Average duration of prepare for global transactions	4.0 and later	Per transaction manager or server	TimeStatistic	Medium
Local before completion duration	transactionModule.localBeforeCompletionDuration	Average duration of before_completion for local transactions	4.0 and later	Per transaction manager or server	TimeStatistic	Medium
Local commit duration	transactionModule.localCommitDuration	Average duration of commit for local transactions	4.0 and later	Per transaction manager or server	TimeStatistic	Medium
Global trans committed	transactionModule.globalTransCommitted	Total number of global transactions committed	3.5.5 and later	Per transaction manager or server	CountStatistic	Low
Global trans rolled back	transactionModule.globalTransRolledBack	Total number of global transactions rolled back	3.5.5 and later	Per transaction manager or server	CountStatistic	Low
Num optimizations	transactionModule.numOptimization	Number of global transactions converted to single phase for optimization	4.0 and later	Per transaction manager or server	CountStatistic	Low
Local trans committed	transactionModule.localTransCommitted	Number of local transactions committed	4.0 and later	Per transaction manager or server	CountStatistic	Low
Local trans rolled back	transactionModule.localTransRolledBack	Number of local transactions rolled back	4.0 and later	Per transaction manager or server	CountStatistic	Low
Global trans timeout	transactionModule.globalTransTimeout	Number of global transactions timed out	4.0 and later	Per transaction manager or server	CountStatistic	Low
Local trans timeout	transactionModule.localTransTimeout	Number of local transactions timed out	4.0 and later	Per transaction manager or server	CountStatistic	Low

Thread pool counters

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Thread creates	threadPoolModule.threadCreates	Total number of threads created	3.5.5 and later	Per thread pool	CountStatistic	Low
Thread destroys	threadPoolModule.threadDestroys	Total number of threads destroyed	3.5.5 and later	Per thread pool	CountStatistic	Low
Active threads	threadPoolModule.activeThreads	The number of concurrently active threads	3.5.5 and later	Per thread pool	RangeStatistic	High
Pool size	threadPoolModule.poolSize	Average number of threads in pool	3.5.5 and later	Per thread pool	BoundedRangeStatistic	High
Percent maxed	threadPoolModule.percentMaxed	Average percent of the time that all threads are in use	3.5.5 and later	Per thread pool	RangeStatistic	High

Web application counters

Data counters for this category contain information for the selected server.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Num loaded servlets	webAppModule.numLoadedServlets	Number of loaded servlets	3.5.5 and later	Per Web application	CountStatistic	Low

Num reloads	webAppModule.numReloads	Number of reloaded servlets	3.5.5 and later	Per Web application	CountStatistic	Low
Total requests	webAppModule.servlets.totalRequests	Total number of requests that a servlet processed	3.5.5 and later	Per servlet	CountStatistic	Low
Concurrent requests	webAppModule.servlets.concurrentRequests	Number of requests that are concurrently processed	3.5.5 and later	Per servlet	RangeStatistic	High
Average response time (ms)	webAppModule.servlets.responseTime	The response time, in milliseconds, of a servlet request	3.5.5 and later	Per servlet	TimeStatistic	Medium
Num errors	webAppModule.servlets.numErrors	Total number of errors in a servlet or JavaServer Page (JSP)	3.5.5 and later	Per servlet	CountStatistic	Low

Workload Management counters

Data counters for this category contain information for workload management.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Num IIOP requests	wlmModule.server.numIncomingRequests	Total number of incoming IIOP requests to an application server	5.0	Per server	CountStatistic	Low
Num strong affinity IIOP requests	wlmModule.server.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

Num no affinity IIOP requests	wlmModule.server.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
Num non-WLM enabled IIOP requests	wlmModule.server.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
Num server cluster updates	wlmModule.server.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
Num of WLM clients serviced	wlmModule.server.numOfWLMClientsServiced	Number of WLM enabled clients that have been serviced by this application server.	5.0	Per server	CountStatistic	Low
Num concurrent requests	wlmModule.server.numOfConcurrentRequests	Number of remote IIOP requests currently being processed by this server	5.0	Per server	RangeStatistic	High

Server response time	wlmModule.server.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
Num outgoing IIOP requests	wlmModule.client.numOfOutgoingRequests	The total number of outgoing IIOP requests being sent from a client to an application server	5.0	Per WLM	CountStatistic	Low
Num server cluster updates	wlmModule.client.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
Client response time	wlmModule.client.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 counters

Data counters for this category contain information for a machine (node).

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level

CPU utilization	systemModule.cpuUtilization	The average system CPU utilization taken over the time interval since the last reading. Because the first call is required to perform initialization, a value such as 0, which is not valid, will be returned. All subsequent calls return the expected value. On SMP machines, the value returned is the utilization averaged over all CPUs.	5.0	Per node	CountStatistic	Low
Free memory	systemModule.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
Average CPU utilization	systemModule.avgCpuUtilization	The average percent CPU Usage that is busy after the server is started	5.0	Per node	TimeStatistic	Medium

Dynamic cache counters

You can use the PMI data for Dynamic Cache to monitor the behavior and performance of the dynamic cache service. For information on the functions and usages of dynamic cache, refer to Using the dynamic cache service to improve performance.

Use the DynaCache MBean to access the related data and display it under Dynamic Cache in TPV.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Max in memory cache size	cacheModule.maxInMemoryCacheSize	Maximum number of in-memory cache entries	5.0	Per server	CountStatistic	Low
In memory cache size	cacheModule.inMemoryCacheSize	Current number of in-memory cache entries	5.0	Per server	CountStatistic	Low
Timeouts	cacheModule.totalTimeoutInvalidations	Aggregate of template timeouts and disk timeouts	5.0	Per server	CountStatistic	Low
Hits in memory	cacheModule.template.hitsInMemory	Requests for this cacheable object served from memory	5.0	Per template	CountStatistic	Low
Hits on disk	cacheModule.template.hitsOnDisk	Requests for this cacheable object served from disk	5.0	Per template	CountStatistic	Low
Explicit invalidations	cacheModule.template.explicitInvalidations	Total explicit invalidation issued for this template	5.0	Per template	CountStatistic	Low

LRU invalidations	cacheModule.template.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
Timeouts	cacheModule.template.timeoutInvalidations	Cache entries evicted from memory or disk, or both, because their timeout has expired	5.0	Per template	CountStatistic	Low
Entries	cacheModule.template.entries	Current number of cache entries created from this template. Refers to the per-template equivalent of totalCacheSize.	5.0	Per template	CountStatistic	Low
Misses	cacheModule.template.misses	Requests for this cacheable object that were not found in the cache	5.0	Per template	CountStatistic	Low
Client requests	cacheModule.template.requestsFromClient	Requests for this cacheable object generated by applications running on the application server	5.0	Per template	CountStatistic	Low
Distributed requests	cacheModule.template.requestsFromJVM	Requests for this cacheable object generated by cooperating caches in this cluster	5.0	Per template	CountStatistic	Low
Explicit invalidations (memory)	cacheModule.template.explicitInvalidationsFromMemory	Explicit invalidations resulting in an entry being removed from memory	5.0	Per template	CountStatistic	Low
Explicit invalidations (disk)	cacheModule.template.explicitInvalidationsFromDisk	Explicit invalidations resulting in an entry being removed from disk	5.0	Per template	CountStatistic	Low
Explicit invalidations (no op)	cacheModule.template.explicitInvalidationsNoOp	Explicit invalidations received for this template where no corresponding entry exists	5.0	Per template	CountStatistic	Low
Local explicit invalidations	cacheModule.template.explicitInvalidationsLocal	Explicit invalidations generated locally, either programmatically or by a cache policy	5.0	Per template	CountStatistic	Low
Remote explicit invalidations	cacheModule.template.explicitInvalidationsRemote	Explicit invalidations received from a cooperating JVM in this cluster	5.0	Per template	CountStatistic	Low
Remote creations	cacheModule.template.remoteCreations	Entries received from cooperating dynamic caches	5.0	Per template	CountStatistic	Low

Web services gateway counters

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

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level

Synchronous requests	wsgwModule.synchronousRequests	Number of synchronous requests made	5.0	Per Web service	CountStatistic	Low
Synchronous responses	wsgwModule.synchronousResponses	Number of synchronous responses made	5.0	Per Web service	CountStatistic	Low
Asynchronous requests	wsgwModule.asynchronousRequests	Number of asynchronous requests made	5.0	Per Web service	CountStatistic	Low
Asynchronous responses	wsgwModule.asynchronousResponses	Number of asynchronous responses made	5.0	Per Web service	CountStatistic	Low

Web services counters

Data counters for this category contain information for the Web services.

Counter definitions

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

Alarm Manager counters

Data counters for this category contain information for the Alarm Manager.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level

Alarms created	Alarms created	Total number of alarms created by all asynchronous scopes for this WorkManager	5.0 and later	Per WorkManager	CountStatistic	High
Alarms cancelled	Alarms Cancelled	Number of alarms cancelled by the application	5.0 and later	Per WorkManager	CountStatistic	High
Alarms fired	Alarms fired	Number of alarms fired	5.0 and later	Per WorkManager	CountStatistic	High
Alarm latency	Alarm latency (ms)	Latency of alarms fired in milliseconds	5.0 and later	Per WorkManager	RangeStatistic	High
Alarms pending	Alarms pending	Number of alarms waiting to fire	5.0 and later	Per WorkManager	RangeStatistic	High
Alarms per sec	Alarms per sec	The number of alarms firing per second	5.0 and later	Per WorkManager	RangeStatistic	High

Object Pool counters

Data counters for this category contain information for Object Pools.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Objects created	Objects created	Total number of objects created	5.0 and later	Per ObjectPool	CountStatistic	High
Objects allocated	Objects allocated	Number of objects requested from the pool	5.0 and later	Per ObjectPool	CountStatistic	High
Objects returned	Objects returned to pool	Number of objects returned to the pool	5.0 and later	Per ObjectPool	CountStatistic	High
Object idle	Idles object in pool	Average number of idle object instances in the pool	5.0 and later	Per ObjectPool	RangeStatistic	High

Scheduler counters

Data counters for this category contain information for the Scheduler service.

Counter definitions

Name	Key	Description	Version	Granularity	Type	Level
Failed tasks	Failed Tasks	Number of tasks that failed to execute	5.0 and later	Per Scheduler	CountStatistic	High
Executed tasks	Executed Tasks	Number of tasks executed successfully	5.0 and later	Per Scheduler	CountStatistic	High
Number of polls	Number of Polls	Number of poll cycles completed for all daemon threads	5.0 and later	Per Scheduler	CountStatistic	High
Tasks per second	Tasks per sec	Number of tasks executed per second	5.0 and later	Per Scheduler	RangeStatistic	High
Collisions per sec	Collisions per sec	Number of collisions encountered per second between competing poll daemons	5.0 and later	Per Scheduler	RangeStatistic	High
Time for poll	Time for poll query (ms)	Execution time in milliseconds for each poll daemon thread's database poll query	5.0 and later	Per Scheduler	RangeStatistic	High
Task execution Time	Task execution Time (ms)	Time in milliseconds taken to execute a task.	5.0 and later	Per Scheduler	RangeStatistic	High
Tasks expiring per poll	Tasks expiring per poll	Number of tasks in a poll query	5.0 and later	Per Scheduler	RangeStatistic	High

Task latency	Task latency (secs)	Period of time in seconds that the task is delayed	5.0 and later	Per Scheduler	RangeStatistic	High
Poll time	Poll time (secs)	Number of seconds between poll cycles	5.0 and later	Per Scheduler	RangeStatistic	High
Tasks executed per poll	Tasks executed per poll	Number of tasks executed by each poll daemon thread. (Multiply this by the number of poll daemon threads to get the tasks executed per effective poll cycle.)	5.0 and later	Per Scheduler	RangeStatistic	High

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.

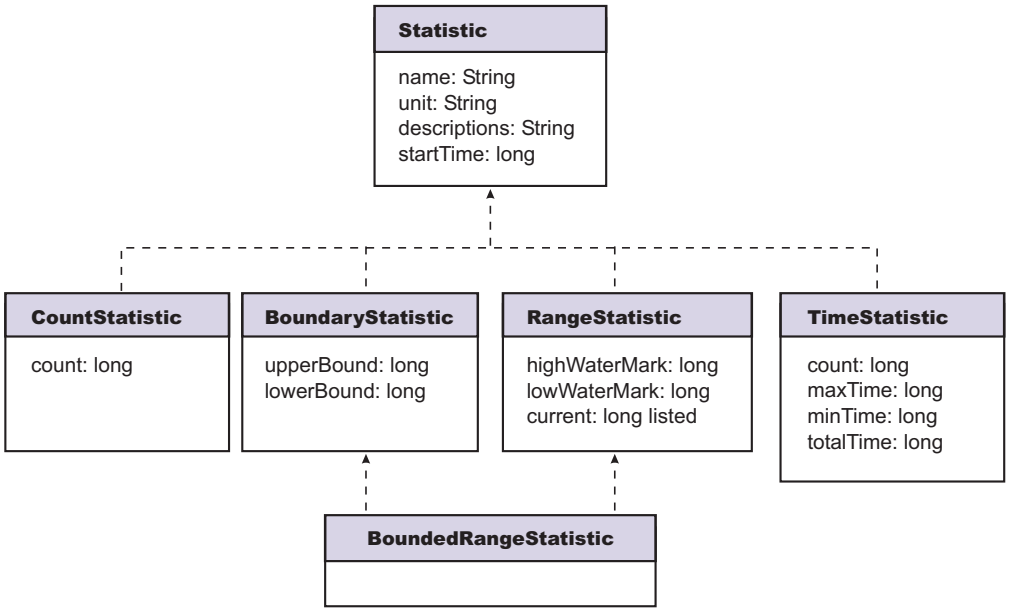
The PMI data can be one of the following statistical types defined in the JSR-077 specification:
- CountStatistic
- BoundaryStatistic
- RangeStatistic
- TimeStatistic
- BoundedRangeStatistic

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.

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.

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

Note: If you are running your monitoring applications with security enabled, refer to Running your monitoring applications with security enabled.

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.

Enabling performance monitoring services in the Node Agent 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.

1. Open the administrative console.
2. Click **System Administration** > **Node Agents** in the console navigation tree.
3. Click *nodeagent*.
4. Click **Performance Monitoring Service**.
5. Select the **Startup** check box .
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 Node Agent. The changes take affect after you restart the Node Agent.

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.

Note: If you are running your monitoring applications with security enabled, refer to Running your monitoring applications with security enabled.

Enabling performance monitoring services using the command line

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

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.

   Note: If PMI data are not enabled yet, you need to first enable PMI data by invoking setInstrumentationLevel operation on PerfMBean.

   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

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 - enable/disable PMI counters

- 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 $perfONname 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 If you know the MBean ObjectName, you can invoke the method by passing the right parameters. 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 $perfONname 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 This operation takes an additional String parameter and it is used for PMI modules that do not have matching MBeans. In this case, the parent MBean is used with a String name representing the PMI module. The String names available in a MBean can be found by invoking listStatMemberNames. For example, beanModule is a logic module aggregating PMI data over all EJBs but there is no MBean for beanModule. Therefore, you can pass server MBean ObjectName and a String ″beanModule″ to get PMI data in beanModule.

- 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 PMI data in all the EJBs within the BeanModule hierarchy since the recursive flag is set to true.

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:

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 **Generic JVM arguments** 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 Java Virtual Machine Profiler Interface (JVMPI) enables the collection of information, such as data about garbage collection, and the Java virtual machine (JVM) API that runs the application server. The Tivoli Performance Viewer leverages a Java Virtual Machine Profiler Interface (JVMPI) to enable more comprehensive performance analysis.

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 and analyzing performance data

WebSphere Application Server performance data, once collected, can be monitored and analyzed with a variety of tools.

1. Monitor performance data with Tivoli Performance Viewer. This tool is included with WebSphere Application Server.
2. Monitor performance data with other Tivoli monitoring tools.
3. Monitor performance data with user-developed monitoring tools. Write your own applications to monitor performance data.
4. Monitor performance with third-party monitoring tools.

Monitoring performance with Tivoli Performance Viewer (formerly Resource Analyzer)

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

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

Performance Viewer can connect to a local or to a remote host. Connecting to a remote host will minimize performance impact to the application server environment.

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

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.

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.

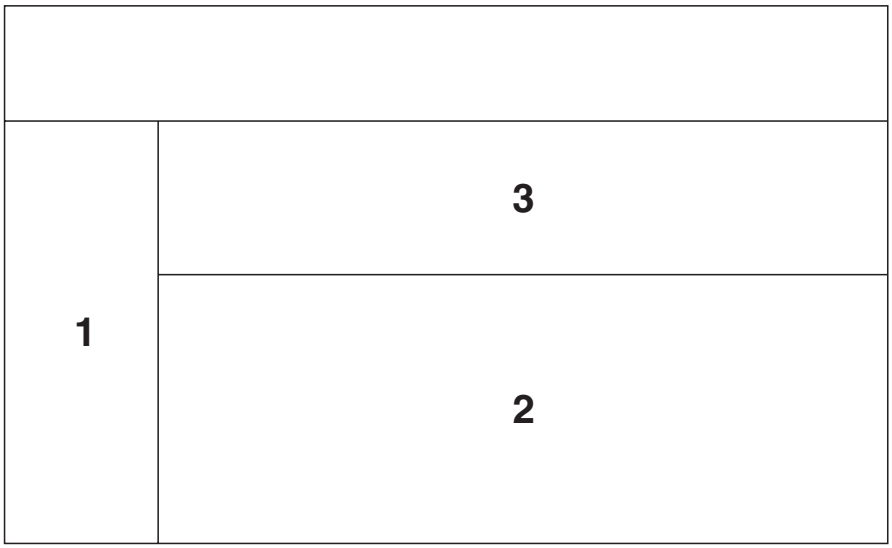
You can do the following tasks with the Tivoli Performance Viewer:

- 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 with the PMI data 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 information center 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 provide 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

You can also start the Tivoli Performance Viewer with security enabled. To do this see Running your monitoring applications with security enabled.

1. Start the Tivoli Performance Viewer. This can be done in two ways:

a. Start performance monitoring from the command line. Go to the *product_installation_directory*/bin directory and run the tperfviewer script.

   You can specify the host and port in Windows NT, 2000, and XP 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. The port numbers for SOAP/RMI connector can be configured in the Administrative Console under **Servers** > **Application Servers** > **server_name** > **End Points**.

   If you are connecting to WebSphere Application Server, use the application server host and connector port. If additional servers have been created, then use the appropriate server port for which data is required. Tivoli Performance Viewer will only display data from one server at a time when connecting to WebSphere Application Server.

   If you are connecting to WebSphere Application Server Network Deployment, use the deployment manager host and connector port. Tivoli Performance Viewer will display data from all the servers in the cell. Tivoli Performance Viewer cannot connect to an individual server in WebSphere Application Server Network Deployment.

   8879 is the default SOAP connector port for WebSphere Application Server Network Deployment.

   8880 is the default SOAP connector port for WebSphere Application Server.

   9809 is the default RMI connector port for WebSphere Application Server Network Deployment.

   2809 is the default RMI connector port for WebSphere Application Server.

   On iSeries, you can connect the Tivoli Performance Viewer to an iSeries instance from either a Windows, an AIX, or a UNIX client as described above. To discover the RMI or SOAP port for the iSeries instance, start Qshell and enter the following command:

   ```
   product_installation_directory/bin/dspwasinst -instance myInstance
   ```

   where
   - *product_installation_directory* is your iSeries install directory
   - *myInstance* is the instance used when you created iSeries instance.

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 SOAP 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.

2. Adjust the data collection settings. Refer to the instructions in the topicSetting performance monitoring levels.

Setting performance monitoring levels

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

To view monitoring settings:

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 with monitoring level set to high
   - **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**.

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 Enabling performance monitoring services in the application server through the administrative console for more information.

Setting monitoring levels for individual enterprise bean methods:

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

To monitor individual methods:

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**.

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

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

The standard monitoring level will enable all reports except the report on EJB methods. To enable EJB methods report, use the custom monitoring setting and set the monitoring level to Max for the Enterprise Beans module.

To view the summary reports:

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

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

To change the rate at which data is retrieved:

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:

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

The **View Chart** tab displays a graph with time as the *x-axis* and the performance value as the *y-axis*.

1. Click a resource in the Resource Selection panel. The Resource Selection panel, located on the left side, 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. See Tivoli Performance Viewer features for information on the Resource Selection panel.
2. Click the **View Chart** tab in the Data Monitoring panel. The Data Monitoring panel, located on the right side, 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. 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 each counter so that the graph allows meaningful comparisons of different counters. Follow these steps to manually adjust the scale:

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.

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.
- 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 variable is graphed as one and one-half times its actual value.
- 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 variable is graphed as one-half its actual value.

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:

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

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:

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

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:

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.

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

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

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

To replay a log file, do the following:

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 path name in the **Log** field.
4. Click **Apply**.
5. Play the log by using the **Play** icon or click **Setting** > **Log Replay** > **Play**.

By default, the data replays at the same rate it was collected or written to the log. 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** > **FF**.

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:

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**.

Developing your own monitoring applications

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

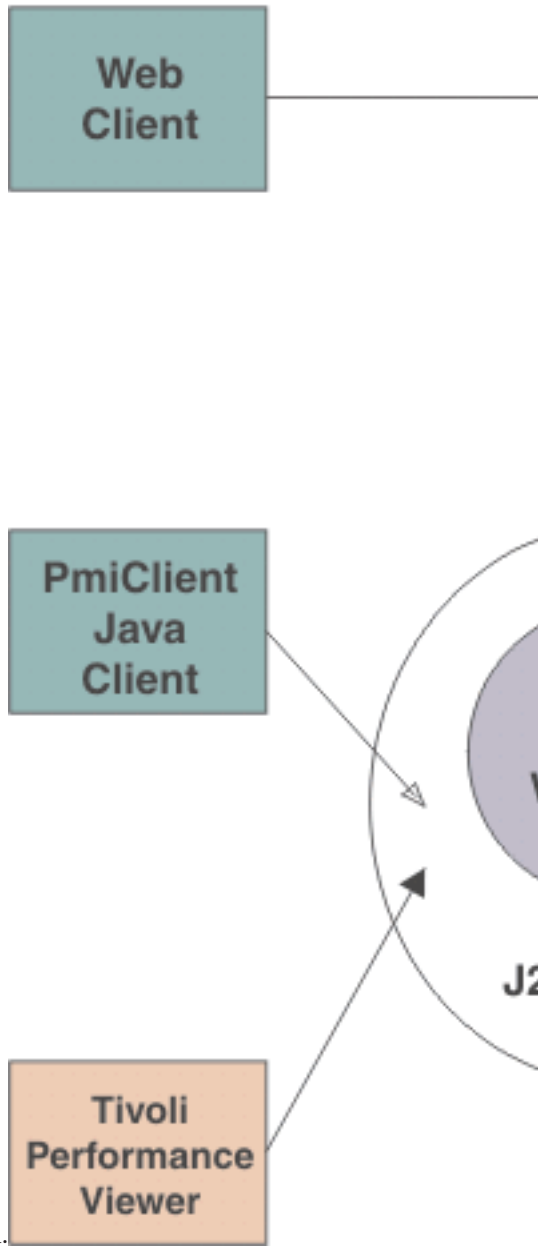
There are three such interfaces - a Java Machine Extension (JMX)-based interface, a PMI client interface, and a servlet interface. All three interfaces return the same underlying data. The JMX interface is accessible through the AdminClient tool. The PMI client interface is a Java interface that works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

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).

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.



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.

Developing your own monitoring application using Performance Monitoring Infrastructure client

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

1. Create an instance of PmiClient. This is used for all subsequent method calls.
2. Optional: You can create your own MBeans. Refer to Extending the WebSphere Application Server administrative system with custom MBeans.
3. 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.
4. Call listMBeans and listStatMembers to get all the available MBeans and MBeanStatDescriptors.
5. Call the getStats method to get the Stats object for the PMI data.

6. Optional: The client can also call setStatLevel or getStatLevel to set and get the monitoring level. Use the MBeanLevelSpec objects to set monitoring levels.

If you prefer to use the Version 4.0 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 (Version 4.0):

A Performance Monitoring Infrastructure (PMI) client is an application that receives PMI data from servers and processes this data.

In Version 4.0, PmiClient API takes PerfDescriptor(s) and returns PMI data as a CpdCollection object. Each CpdCollection could contain a list of CpdData, which has a CpdValue of the following types:

- CpdLong
- CpdStat
- CpdLoad

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:

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")
   ```

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
            // 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
              nodeName = nodePds[0].getName();
            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

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:

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`

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>
  ```

Developing your own monitoring application with the Java Management Extension interface

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:

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:

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 ir
            //--------------------------------------------------------------------------
            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 cos
        //       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="

        // 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, "getInstrumentationLeve
        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, "getInstrumentationLeve
        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(PmiClient.findConfig(configs, 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(PmiClient.findConfig(configs, 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 + "    ");
    }
  }


/**
 * The Stats object returned from server does not have static config info. You have to set it on
 */
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 = PmiClient.findConfig(configs, 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", param
        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", param
        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", param
        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,
        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(jvmONname,
                                                                    "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(PmiClient.findConfig(configs, jvmONname));

            // 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)

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:

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.

Compiling your monitoring applications

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
- ras.jar
- wasjmx.jar
- j2ee.jar
- soap.jar
- soap-sec.jar
- nls.jar
- ws-config-common.jar
- namingclient.jar

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

Running your new monitoring applications

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.

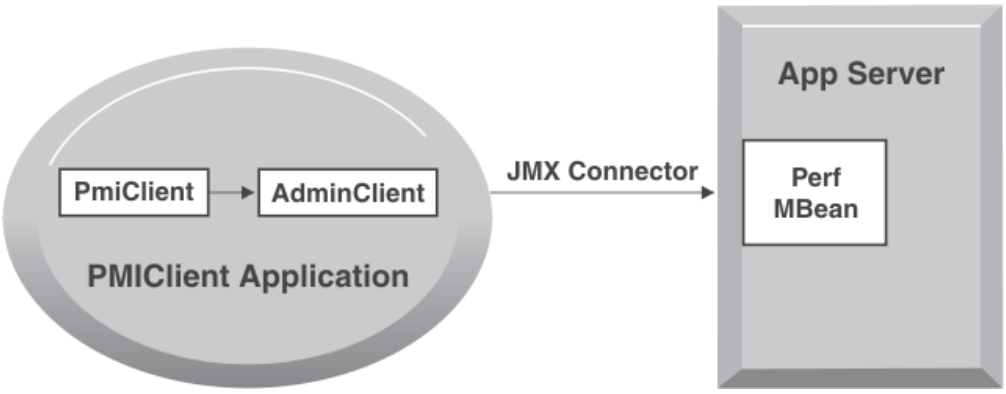
As shown in the following figure, PmiClient uses the AdminClient API to communicate the Perf MBean in an application server.

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.

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.

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 information center 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″.

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 is a tool that allows you to track individual transactions, recording the processing time in each of the major WebSphere Application Server components. The information tracked may either be saved to log files for later retrieval and analysis, be sent to ARM Agents, or both.

As a transaction flows through the system, Request Metrics tacks on additional information so that the log records from each component can be correlated, building up a complete picture of that transaction. The result looks similar to the following:

```
HTTP request /trade/scenario               ------------------------------> 172 ms
     Servlet/trade/scenario             ---------------------------------> 130 ms
         EJB   TradeEJB.getAccountData                                 -->  38 ms
              JDBC select                                          ->   7 ms
```

This transaction flow with associated response times can help users target performance problem areas and debug resource constraint problems. For example, the flow can help determine if a transaction is spending most of its time in the Web server plug-in, the Web container, the enterprise bean container or the backend database. The response time collected for each level includes the time spent at that level and the time spent in the lower levels. For example, the response time for the servlet, which is 130 milliseconds, also includes 38 milliseconds from the EJB and JDBC. Therefore, 92ms can be attributed to the servlet process.

Request metrics tracks the response time for a desired transaction. For example, tools can inject synthetic transactions. Request Metrics can then track the response time within the WebSphere environment for those transactions. A synthetic transaction is one that is injected into the system by administrators in order to proactively test the performance of the system. This information can help administrators tune the performance of the website and take corrective actions should they be needed. Thus, the information provided by Request Metrics might be used as an alert mechanism to detect when the performance of particular request type goes beyond acceptable thresholds. The filtering mechanism within Request Metrics may be used to focus on the specific synthetic transactions and can help optimize performance in this scenario.

Three types of filters are supported:
- Originator IP filter
- URI filter
- EJB method name filter

When filtering is enabled, only requests matching the filter generate Request Metrics data, create log records, and/or call the ARM interfaces. This allows work to be injected into a running system (specifically to generate trace information) to evaluate the performance of specific types of requests in the context of normal load, ignoring requests from other sources that might be hitting the system.

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

Typically, there are multiple processes across several nodes in a distributed system. When a request comes to a process, the process may send the request to one or more downstream processes.

Trace records may be generated for each process with associated elapsed times for that process. These trace records may be correlated together to build a complete picture of the request flow through the distributed system, similar to the diagram in Measuring data requests (Performance Monitoring Infrastructure Request Metrics).

The process response time monitored by Request Metrics can be viewed through the Application Response Measurement (ARM) interface and system logs. For requests that originate from either an HTTP request or the remote interface of an enterprise bean, Request Metrics captures response times for the initiating request and any related downstream invocations. If the request originated as an HTTP request, response times are generated for web server plug-in (only available when using web server port), the web container, the EJB container, and JDBC calls. If the request originated as a remote EJB call, response times are generated for the EJB container and JDBC calls. Note that the JDBC response time are only traced for the WebSphere 5.0 data source. No response time will be traced for WebSphere 4.0 data source.Depending on the trace level configured for request metrics, the intra-process servlet and EJB calls may or may not be traced.

When active, Request Metrics compares each incoming request to a set of known filters. Three types of filters are supported:

- Originator IP filter
- URI filter
- EJB method name filter

When filtering is enabled, only requests matching the filter generate Request Metrics data, create log records, and/or call the ARM interfaces. This allows work to be injected into a running system (specifically to generate trace information) to evaluate the performance of specific types of requests in the context of normal load, ignoring requests from other sources that might be hitting the system. If the request matches any filter with a trace level greater than TRACE_NONE, trace records are generated for that request.

Trace records are generated and logged for the Web Server plug-in, servlets (WebContainer), remote enterprise bean calls, and Java Database Connectivity (JDBC drivers).

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. One product that uses ARM agents is described in Tivoli performance monitoring and management solutions.ARM 4.0 agent is supported beginning with Version 5.1.1 and later.

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 or sent through ARM for real-time and historical analysis.

Incoming HTTP requests

HTTP requests arriving at the WebSphere Application Server may be filtered based on the URI and/or the IP address of the originator of the request.

- **Originator 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 to be 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 Originator IP address filters.
- **Filter combinations.**If both URI and Originator 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, the asterisk (*) may be used in the mask. If used, the asterisk must always be the last character of a filter pattern.

Because the ability to track the request response times comes with a cost, filtering helps optimize performance when using Request Metrics.

Performance Monitoring Infrastructure Request Metrics data output

The trace records for Performance Monitoring Infrastructure (PMI) Request Metrics data are output to two log files: the web server plug-in log file and the application server log file. The default directory for these log files is <$WAS_ROOT/logs/server1> (or the name given to your server <$WAS-ROOT/logs/*server_name*>) and default names are SystemOut.log and http_plugin.log. Users may, however, specify these log file names and their locations.

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
```

In the Web server plug-in log file the trace record format is:

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

The trace record format 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. Refer to Measuring data requests (Performance Monitoring Infrastructure Request Metrics) for an example of the transaction flow.

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 `HTTP`, `URI`, `EJB` and `JDBC`.
- **detail**: Identifies the name of the operation being timed (See the description of Universal Resource Identifier (URI), `HTTP`, 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.
- **bytesIn**: The number of bytes from the request received by the Web server plug-in.
- **bytesOut**: The number of bytes from the reply sent from the Web server plug-in to the client.

The type and detail fields are described as follows:
- **HTTP**: The Web server plug-in generates the trace record. The detail is the name of the URI used to invoke the request.
- **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 **Troubleshooting** > **PMI Request Metrics** from the administrative console navigation tree.

Tasks included in configuring Request Metrics:

1. Enable Request Metrics.
2. Enable Application Response Measurement (ARM).
3. Enable Request Metrics filters.
4. Add and remove Request Metrics filters.
5. Set the trace level in Request Metrics.
6. Update the Web server plug-in configuration file.

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 - reserved for enabling additional information over hops, but is not as performance intensive as DEBUG; DEBUG - reserved for full detailed trace. However, currently both PERF_DEBUG and DEBUG provide the same level of performance tracing as the HOPS level.

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. Note: this has to be enabled in order to enable the filter values under this filter type.

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.

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.

1. Open the administrative console.
2. Click **Troubleshooting** > **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**.

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

Enabling Application Response Measurement

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 the ARM agent is supported with the request metrics in WebSphere.

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.

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 **Troubleshooting** > **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**.

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.

1. Open the administrative console.
2. Click **Troubleshooting** > **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.

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:

1. Open the administrative console.
2. Click **Problem Determination** > **PMI Request Metrics** in the console navigation tree.
3. Click **filters**.
4. Choose a filter type.
   a. Click on filterValues.

b. You can edit, add, and delete a filter value. To edit, click on a filter value and change its value and enablement. To add, click on ″New″ and type in the value and optionally check the ″Enable″ box. To delete, select a filter value and click on ″Delete″.

5. Click **Apply** or **OK**.
6. 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 in the IPv4 addressing format.

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:

1. Open the administrative console.
2. Click **Troubleshooting** > **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**.

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

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.

1. Open the administrative console.
2. Regenerate the Web server plug-in configuration.

Example: Generating trace records from Performance Monitoring Infrastructure Request Metrics

Use `HitCount` enterprise bean `/hitcount?selection=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/hitcount?selection=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/hitcount?selection=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=/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=/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.defaultapplication.ConcreteIncrement_501bb55e.increment elapsed=40
```

Note: The detail for an EJB call is close to the class name of the EJB but not exactly the class name.

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)

  The Open Group ARM specifications.

Monitoring performance with third-party tools
- Enterprise Web Application Management 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
- 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

  See WebSphere Application Server Development Best Practices for Performance and Scalability for more information on application tuning.
- Performance Analysis for Java Web sites
- AIX documentation

  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

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

  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 (Version 4.02)
- Redbook: WebSphere Application Server V3.5 Handbook (SG24-6161-00)
- Redbook: WebSphere Application Server V3 Performance Tuning Guide (SG24-5657-00)

Garbage collection
- IBM developerWorks

  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

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

Chapter 3. Using the Runtime Performance Advisor

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.

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. If there are specific counters that are not wanted, or when disabling the Runtime Performance Advisor, the user may want to disable the PMI Service or the counters that RPA enabled.
2. Start the administrative console.
3. Click **Servers** > **Application Servers** in the console navigation tree.
4. Click *server_name* > **Runtime Performance Advisor Configuration**.
5. Click the **Configuration** tab.
6. Select the **Number of Processors**. Select the appropriate settings for your system configuration to ensure accurate advice.
7. Select the **Calculation Interval**. PMI data is taken over an interval of time and averaged to provide advice. The calculation 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.
8. 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.
9. Click **Apply**.
10. Click **Save**.
11. Click the **Runtime** tab.
12. 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.

13. 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.
14. Select the check box to enable the Runtime Performance Advisor.

    Note: To achieve the best results for performance tuning, enable the Runtime Performance Advisor when a stable production level load is being applied.
15. Click **OK**.
16. 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.
17. 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.

    Over a period of time the advisor may issue differing advice. This is due to load fluctuations and runtime state. When differing advice is received the user should look at all advice and the time period over which it was issued. Advice should be taken during the time that most closely represents peak production load.

    Performance tuning is an iterative process. After applying advice, simulate a production load, update the configuration based on the advice, and retest for improved performance. This procedure should be continued until optimal performance is achieved.

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

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

For more information on how to use the Runtime Performance Advisor, see Using 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 will run on server startup.

The Runtime Performance Advisor (RPA) requires that the Performance Monitoring Service be enabled. It does not require that individual counters be enabled. When a counter that is needed by RPA is not enabled, RPA will enable it automatically. When disabling the Runtime Performance Advisor the user may want to disable the PMI Service or the counters that RPA enabled. The following counters may be enabled by RPA:

- ThreadPools (module)
  - Web Container (module)
    - Pool Size
    - Active Threads
  - Object Request Broker (module)
    - Pool Size
    - Active Threads
- JDBC Connection Pools (module)
  - Pool Size
  - Percent used
  - Prepared Statement Discards
- Servlet Session Manager (module)
  - External Read Size
  - External Write Size
  - External Read Time
  - External Write Time
  - No Room For New Session
- System Data (module)
  - CPU Utilization
  - Free Memory

Enable Runtime Performance Advisor

Starts the Runtime Performance Advisor on the current server.

The Runtime Performance Advisor (RPA) requires that the Performance Monitoring Service be enabled. It does not require that individual counters be enabled. When a counter that is needed by RPA is not enabled, RPA will enable it automatically. When disabling the Runtime Performance Advisor the user may want to disable the PMI Service or the counters that RPA enabled. The following counters may be enabled by RPA:

- ThreadPools (module)
  - Web Container (module)
    - Pool Size
    - Active Threads
  - Object Request Broker (module)
    - Pool Size
    - Active Threads
- JDBC Connection Pools (module)
  - Pool Size
  - Percent used
  - Prepared Statement Discards
- Servlet Session Manager (module)
  - External Read Size

  - External Write Size
  - External Read Time
  - External Write Time
  - No Room For New Session
- System Data (module)
  - CPU Utilization
  - Free Memory

Calculation Interval

PMI data is taken over an interval of time and averaged to provide advice. The calculation interval specifies the length of the time over which data is taken for this advice. Details within the advice messages will appear as averages over this interval.

Maximum warning sequence

The maximum warning sequence refers to the number of consecutive warnings issued before the threshold is relaxed.

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.

Number of processors

Specifies the number of processors on the server.

This setting is critical to ensure accurate advice for the system's specific configuration.

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 sequence is reset to zero for each recommendation/advice.

Advice configuration settings

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).

Chapter 4. Using the Performance Advisor in Tivoli Performance Viewer

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.

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.
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 performance monitoring services through Tivoli Performance Viewer.
   b. 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. 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.

Note: If the Refresh Rate is adjusted, the Buffer Size should also be adjusted to allow sufficient data to be collected for performing average calculations. Currently 2 minutes of data is required. Read more about adjusting the Refresh Rate or Buffer Size at:
- Changing the refresh rate of data retrieval.
- Changing the display buffer size.

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.

   Over a period of time the advisor may issue differing advice. This is due to load fluctuations and runtime state. When differing advice is received the user should look at all advice and the time period over which it was issued. Advice should be taken during the time that most closely represents peak production load.

   Performance tuning is an iterative process. After applying advice, simulate a production load, update the configuration based on the advice, and retest for improved performance. This procedure should be continued until optimal performance is achieved.
10. Clear values from tables and charts.
11. Reset counters to zero.

Performance Advisor Report in Tivoli Performance Viewer

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.

For more information on how to use the Performance Advisor in Tivoli Performance Viewer effectively, see Using the Performance Advisor in Tivoli Performance Viewer.

Message

Specifies recommendations for performance tuning.

Double click the message to obtain more details.

Performance data in lower panel

Displays a summary of performance data for the WebSphere Application Server. Data here corresponds to the same period that recommendations were provided for. However, recommendations may use a different set of data points during analysis than the set displayed by the summary page.

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 CPU activity as percentage busy and idle.

The bar graphs display average thread activity for the Web container and Object Request Broker (ORB) thread pools, and average database connection activity for connection pools. Activity is expressed as the number of threads/connections busy and idle.

Chapter 5. Tuning performance parameter index

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 performance 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.

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.

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.

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.

- Business process choreographer
- Business Rule Beans
- Dynamic query service
- Object pool
- Work area
- Asynchronous beans
- ActivitySession
- Application profiling

Business process choreographer You can use the process choreographer to implement businesses processes. Business process management systems support the definition and execution of business processes or flows. Review the Process choreographer tuning tips for best practices.

- **Process message-driven beans listener port maximum sessions**
  - **Description:** Specifies the maximum number of concurrent JMS server sessions used by a listener to process messages. Adjust this parameter when the machine running the process application does not realize the available capacity and produces less throughput during long running processes.
  - **How to view or set:** Set the maximum sessions through the `server.xml` file or through the administrative console. In the `server.xml` file, change the *Max sessions* value as specified under listener ports stanza:

```
<listenerPorts xmi:id="ListenerPort_1" name="bpeIntListenerPort" description="Internal Listener
<stateManagement xmi:id="StateManageable_5" initialState="START"/>
</listenerPorts>
```

You can set this parameter through the administrative console by Configuring a listener port.

- **Default value:** 5
- **Recommended value:**

  A value of 25 has given the best throughput with the above configuration. Based on the amount of work and the resources available, set a value between 15 and 25 to obtain the maximum process throughput. By adjusting this parameter to the recommended value, the maximum number of concurrent JMS server sessions used by the process message-driven beans increases, increasing message processing capacity and application throughput. A 40% increase in throughput occurs in a process application with long running processes on a NetFinity 5500 500 MHz, 4-way, 4GB RAM system.

Business Rule Beans

Business Rule Beans (BRBeans) are used to create and modify rules that keep pace with complex business practices. Business Rule Beans enable your application's core behavior and user interface objects to remain intact and untouched, even as business practices change. You can improve performance by adjusting the following parameter:

- Rule properties **Firing location**

To learn more about how using BRBeans can improve performance through caching, using indexes and changing the fire location, see BRBeans performance enhancements.

Dynamic query service

You can use the dynamic query service to build and execute queries against entity beans constructed dynamically at runtime, rather than defining them at deployment time. See Dynamic query service performance considerations for information about how dynamic query can improve performance.

Object pool

An object pool enables an application to avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused at a later point. An object pool allows an object to be pooled while waiting for the point when it can be reused. These object pools are not meant to be used for pooling JDBC connections or JMS connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kits, Java technology editions types.

Review suggestions on how object pools can improve performance in Object pool performance considerations.

WorkArea service

The WorkArea service is designed to address complex data passing patterns that can quickly grow beyond convenient maintenance. A work area is in essence a scratchpad that is accessible to any client capable of looking up Java Naming and

Directory Interface (JNDI). Once a work area is established, data can be placed on it for future use in any subsequent method calls, to both remote and local resources.

Review suggestions on how work areas can improve performance in WorkArea service performance considerations.

Asynchronous beans

An asynchronous bean is a Java object or enterprise bean that can be executed asynchronously by a J2EE application, using the J2EE context of the bean's creator. Asynchronous beans can improve performance by enabling a J2EE program to decompose operations into parallel tasks. Asynchronous beans enable the construction of stateful, "active" J2EE applications. These applications address a segment of the application space that J2EE has not previously addressed (that is, advanced applications that require application threading, active agents within a server application, or distributed monitoring capabilities).

Use the following parameters to tune asynchronous beans:
- **Work manager**
  - **Number of alarm threads**
  - **Minimum number of threads**
  - **Maximum number of threads**
  - **Thread priority**
  - **Growable**
  - **Service names**
- **Work manager service**

  By default, the work manager service is enabled at startup. You can enable or disable the work manager service. The overhead for the service is minimal and should not pose a problem if left enabled.
  - **Startup**

For additional topic information see Asynchronous beans.

ActivitySession
- **Description:** Provides an alternative unit-of-work (UOW) scope to that provided by global transaction contexts. An ActivitySession context can live longer than a global transaction context, and can encapsulate global transactions. Disabling the service provides slight performance improvement for remote requests.
- **How to view or set:** In the administrative console, disable or enable the ActivitySession service.
- **Default value:** Enabled
- **Recommended value:** Disable this parameter if applications are not using the ActivitySession service, either through the UserActivitySession API or through deployment descriptors, and the ActivitySession samples are not installed.

Application profiling

Application profiling enables assembly configuration techniques that improve your application runtime, performance and scalability. You can configure tasks that identify incoming requests, identify access intents determining concurrency and other data access characteristics, and profiles that map the tasks to the access intents. The capability to configure the application server can improve performance, efficiency and scalability, while reducing development and maintenance costs.

Review suggestions on how application profiling can improve performance in Application profiling performance considerations.

For more information about the service see Application profiling: Overview.

Tuning hardware capacity and settings

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

- **Optimize 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. Placing database tables on disks that are separate from the disks used for the database log files can reduce disk contention and improve throughput.
- **Increase processor speed and processor cache**
  - **Description:** In the absence of other bottlenecks, increasing the processor speed often helps throughput and response times. A processor with a larger L2 or L3 cache can yield higher throughput even if the processor speed is the same as a CPU with a smaller L2 or L3 cache.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value:** None
- **Increase 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 application server
- **Run network cards and network switches at full duplex.**
  - **Description:** Running at half duplex decreases performance. Verify that the network speed of adapters, cables, switches, and other devices can accommodate the required throughput.
  - **How to view or set:** None
  - **Default value:** None
  - **Recommended value:** Make sure that the highest speed is in use on 10/100/1000 Ethernet networks.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA