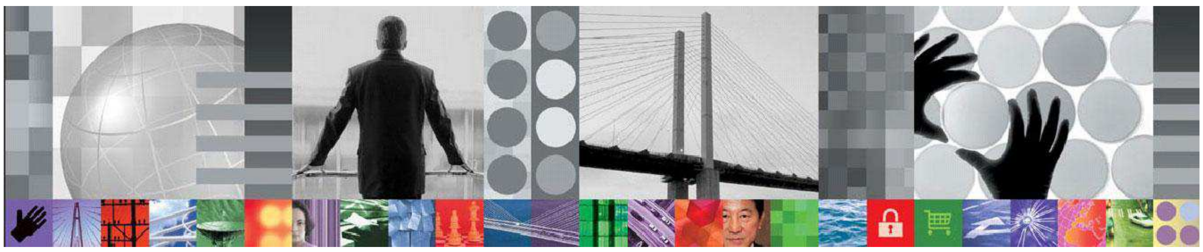
A High Performance SIP and HTTP Web Application Benchmark and its Tuning

Dec 2009



Curtis Hrischuk, Ph.D.

Table of Contents

1 Performance of communication enabled applications ... 3
2 The web voice mail application ... 4
3 System configuration and performance of a single node ... 5
4 System configuration and performance of highly available cluster ... 8
5 HTTP and SIP Performance Differences ... 11
6 Tuning Goals for SIP and HTTP ... 14
7 Tuning a stand-alone WebSphere application server ... 17
7.1 Operating system tuning ... 17
7.2 JVM tuning ... 18
7.2.1 Selecting the garbage collection algorithm ... 18
7.2.2 Determining the size of the Java heap ... 19
7.2.3 Partitioning the Java heap ... 19
7.2.4 JVM settings to avoid compactions ... 21
7.2.5 Additional JVM settings ... 22
7.2.6 Setting the JVM options ... 23
7.3 Application server tuning ... 24
7.4 Tuning the application ... 25
8 Tuning a highly available cluster ... 26
8.1 WebSphere proxy server tuning ... 27
8.2 High availability WebSphere application server tuning ... 28
9 Summary ... 30
10 Appendix: Tuning Information ... 31
11 Resources ... 33

1 Performance of communication enabled applications

Enterprises are adding communications capabilities into new and existing web applications to help customers, partners, and employees interact more efficiently. Enhancing personal communications can increase revenue while reducing costs: social networking can identify more sales leads, self-service helps customers to support themselves, and instant messaging can speed-up business processes.

Many enterprises start down the path of communications enabling their existing applications by adding "Click to Call" capabilities. "Click to Call" is a feature where a user clicks on an HTML button and start a voice conversation to, or from, a physical phone, a mobile phone or a Voice over IP (VoIP) softphone. Communications enabling web applications enables personal communication to occur over the Internet, in real-time.

Java^{TM} Enterprise Edition (Java EE) technologies make it easy to add communications capabilities using Session Initiation Protocol (SIP) programming standards such as JSR 116 [JSR116] or JSR 289 [JSR289]. SIP servlet programming is used to exploit all of the capabilities of the converged HTTP and SIP Servlet container in the IBM® WebSphere® Application Server. WebSphere Application Server easily handles this cutting edge functionality by delivering over 3 million converged operations per hour, running on a single IBM BladeCenter® HS21 blade server. Another development approach is to use a multi-modal development framework to simplify the effort, like the IBM WebSphere Application Server V7 Feature Pack for Communications Enabled Applications (CEA).

Any system benefits from performance tuning, and this is true of a communications enabled application. Many standard HTTP web and Java application server tuning techniques apply. Some additional tuning techniques are needed because SIP has more stringent Quality of Service (QoS) goals. This article describes best practices for tuning a new benchmark that accesses voice mail over the web, combining HTTP and SIP. The benchmark simulates people using a browser to establish a Voice over IP (VoIP) phone call and interact with a voice mail server. Note that the technologies utilized in this benchmark are not limited to VoIP phone calls, and could equally connect calls over a standard landline phone or mobile phone.

This benchmark demonstrates the carrier-grade performance of IBM WebSphere Application Server. The benchmark capacity is measured using a new metric of *Converged Operations per Hour* which is the sustained hourly rate at which new users start and complete the web voice mail application. A converged operation begins with the initial HTTP request and stops with the call ending after an average duration of 60 seconds. An average of 2.5 HTTP GET requests and 28 SIP messages are handled per user. The IBM WebSphere Application Server achieved 3,351,600 Converged Operations per Hour (931 converged operations per second) on a single IBM BladeCenter HS21 Series x® blade server: 2 CPU, quad core machine, running at 3.33GHz with 16 GBytes of RAM. This was achieved with an average CPU utilization of 65%. This is equivalent to 7,218,830 Busy Hour Call Attempts (assuming each call is 13 SIP messages) while also processing 8,379,000 HTTP requests per hour.

This article presents best practices for tuning a converged SIP and HTTP Java EE application, reviewing the most commonly used parameters. Table 3, in the appendix, summarizes the tuning information that is discussed, suggesting initial values that should work well in most circumstances.

The structure of this article moves from high-level, architectural principles to best practice performance tuning procedures, using the example web voice mail benchmark to present measurement data. To provide the context for this material, the performance results are presented along with the configuration used for the measurements. Prior to this, a brief overview of the web voice mail application is given. This is followed by a brief comparison between the HTTP and SIP protocols and the impact on performance. Then the goals that guide the tuning process are identified. Readers new to SIP are referred to [CMG-Sip] for a tutorial on SIP performance.

2 The web voice mail application

The benchmark scenario is quite simple, converting an HTML button click into a voice conversation. The scenario begins with a sales person at a coffee shop wanting to access their voicemail. They access the corporate network, bring up their personal page, and then click on a button to access their voice mail. The button sends an HTTP request to establish a SIP based, voice three way call between: the voice mail system, the user's soft-phone on their netbook, and a WebSphere Application Server instance running the web application. The WebSphere Application Server instance manages the session start and end; the voice traffic is assumed to be carried on a separate path as is customary for SIP. When the user is done they either: (1) hang up the soft-phone (terminate the call by SIP) or (2) click a button on the web page (end the call using HTML). The benchmark roughly equalizes the termination types (i.e., 50% hang up and 50% terminate via the web page). For simplicity, it is assumed that the number of sessions that time-out is negligible.

This scenario translates into many SIP messages and several HTTP requests, as shown in Figure 1. The HTTP requests are shown on the left of the application server and the SIP message exchanges are shown on the right.[1] The voice mail server and application server exchange SUBSCRIBE and NOTIFY events,[2] to determine how many voice mail messages are pending for the user. To keep things simple, the benchmark assumes that there are always four pending voice mail messages when the user initially connects. The average time the user interacts with the voice mail system (i.e., call hold time) is 60 seconds.

The system configuration is a three-tier model. Tier one is the load driver that simulates the activities of Web users. Tier two comprises multiple application servers that receive the requests. Unlike typical web applications where the third tier is a database, the third tier for this application consists of SIP end points that behave like the end user and the voice mail server. From a workload perspective, each user generates 2 or 3 web page requests, as well as 28 SIP messages which are not an uncommon amount of SIP activity. An interesting aspect of the

[1] The soft phone and voice mail server behave according to the best practices in rfc3725: Best Current Practices for Third Party Call Control (3pcc)

[2] This is similar to rfc3842: Message Waiting Indication Event Package

message flows in Figure 1 is that much of the SIP processing occurs in parallel between the various SIP test driver end points.

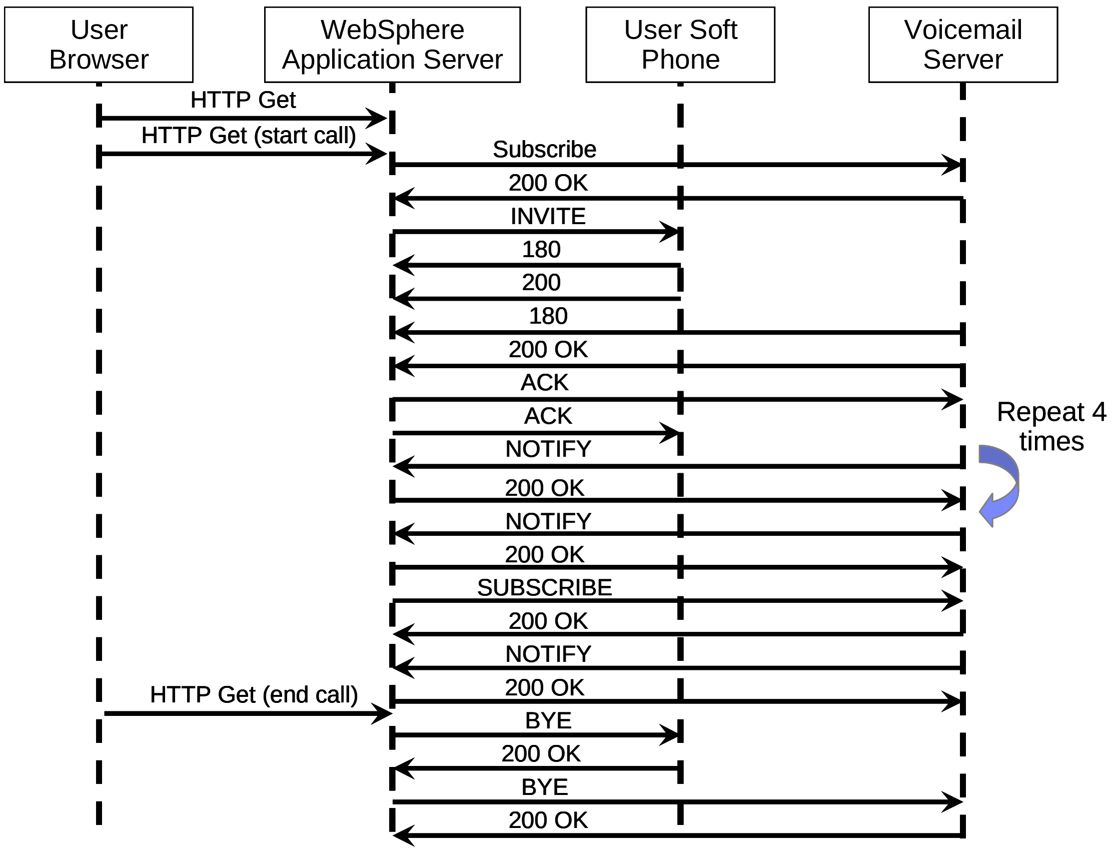
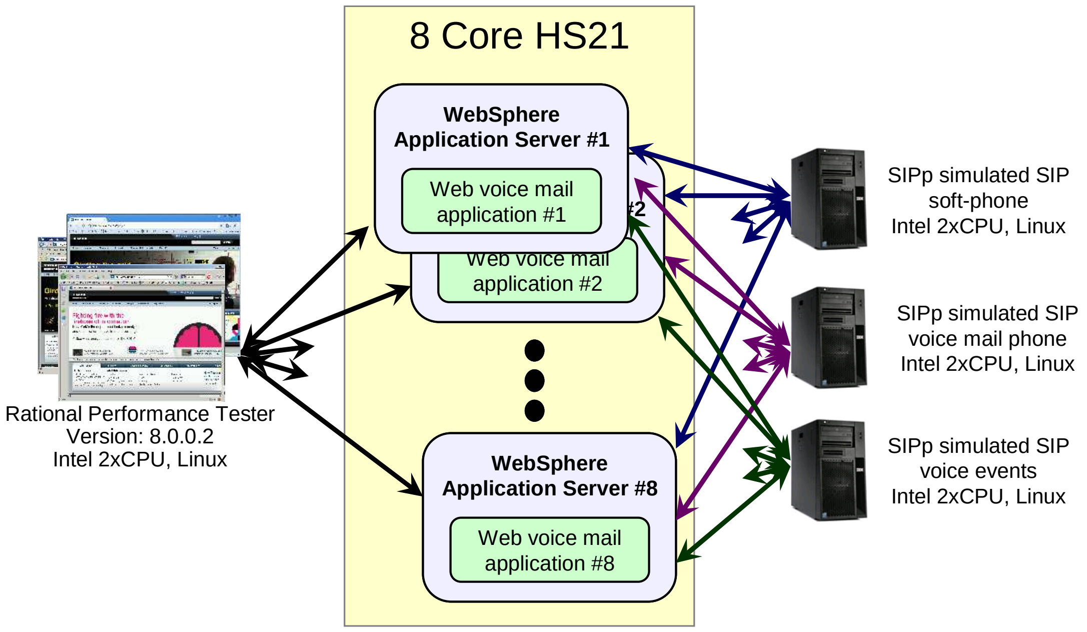


Figure 1: HTTP and SIP Message Exchanges for Ending the Call by a Web Request

3 System configuration and performance of a single node

It is useful to examine the performance characteristics of the System Under Test (SUT) before diving into the details around performance best practices. A standalone node configuration is presented first, followed by a WebSphere Network Deployment configuration (i.e., a cluster) since it builds on the standalone node concepts.

The benchmark configuration is shown in Figure 2. There are eight stand-alone WebSphere Application Servers instances configured on a single hardware node. This technique is referred to as *vertical scaling* because it can be thought of as having multiple JVMs stacked on the single hardware node. Vertical scaling is a standard technique for application servers to more fully utilize hardware resources and increases overall capacity. In particular, vertical scaling helps SIP applications by increasing the total amount of network buffering available to reduce the possibility of messages being lost (this is discussed further in Section 6). Vertical scaling also provides additional fault tolerance in the event a JVM fails. The application server node is an IBM BladeCenter® HS21 xSeries blade server with two 3.33GHz Intel® quad core X5470 CPUs with 16 GBytes of RAM, running Red Hat Enterprise Linux release 5.2.



8 Core HS21
WebSphere Application Server #1
Web voice mail application #1
#2
Web voice mail application #2
WebSphere Application Server #8
Web voice mail application #8
Rational Performance Tester Version: 8.0.0.2 Intel 2xCPU, Linux
SIPp simulated SIP soft-phone Intel 2xCPU, Linux
SIPp simulated SIP voice mail phone Intel 2xCPU, Linux
SIPp simulated SIP voice events Intel 2xCPU, Linux

Figure 2: Software Configuration of a Single Hardware Server

IBM Rational® Performance Tester generates the HTTP load that simulates users accessing their voice mail over the web. This workload is generated using one workbench that manages sixteen load generation machines, each of which executes a single load driving agent. The application servers interact with the SIP load drivers that emulate the user's soft-phone and voice mail server.

The traffic generator program SIPp [SIPp] is used to simulate the SIP end-points of Figure 2. Three SIPp scripts are used: user's soft-phone emulation (two instances), the voice mail VoIP line emulation (two instances), and a voice mail server's event notification emulation (four instances). The SIPp program instances are distributed across several machines to ensure there was ample load generation capacity

The performance results of Figure 3 represent the combined throughput of the eight stand-alone Java EE WebSphere Application Servers running on the single IBM BladeCenter® HS21 xSeries blade server.

The peak capacity is calculated per node which is the aggregate of the eight WebSphere Application Server instances. It is determined by increasing the rate of initial web voice mail HTTP requests until either an HTTP request timed out or one of the SIP interactions had an unexplained failure.[3] At that peak capacity the user request rate, HTTP page request rate, SIP message rate, CPU utilization, and aggregate used application memory are recorded. A Quality of Service latency value is also recorded, noting the delay between a SIP request and its

[3] For the sake of efficiency, the SIPp traffic generator does not fully implement a SIP protocol stack which can result in SIPp reporting a protocol violation when the interaction was valid. This usually is the result of messages being retransmitted or messages being received out of order, which is allowed in the SIP protocol but is falsely reported by SIPp as an error. The SIPp XML script files that characterize the message interactions do take into account some of these issues but, in some cases, manual inspection is required. SIPp does record enough information about each failure so that a manual inspection can determine if an error really occurred.

acknowledgements ("95th Percentile latency from the NOTIFY to 200 acknowledgement message" measurement in Figure 3). This delay is calculated from data recorded by the SIPp load driver, using a 10 minute measurement interval at a fixed call rate.

The peak capacity of the web voice mail application is shown as the "*Converged Operations per Hour*" metric in Figure 3. Since the new benchmark combines HTTP and SIP, a new capacity metric is used: *Converged Operations per Hour*.[4] This metric represents the sustained hourly rate at which new users start and complete the web voice mail application. It begins with the initial HTTP GET request and completes with the softphone hanging up or an HTTP GET that closes down the call. Per user it includes an average of 2.5 HTTP GET requests and 28 SIP messages.

The peak capacity for the stand-alone configuration is 3,351,600 Converged Operations per Hour (931 converged operations per second * 3600 seconds per hour).

The converged operations per hour metric can be broken down into the individual message rates driven by the workload. The HTTP request rate is 2,327 HTTP requests per second (roughly 931 converged operations per second * 2.5 HTTP requests per converged operation), while the SIP request rate is 26,068 SIP messages per second (931 converged operations per second * 28 SIP messages per converged operation). This combined throughput was achieved with an average 65% CPU utilization, allowing room for more advanced business logic without impacting the overall capacity. This is equivalent to 7,218,830 Busy Hour Call Attempts (assuming each call is 13 SIP messages) while also processing 8,379,000 HTTP requests per hour, on a single machine running the same application.

In addition to peak capacity, memory consumption is an important factor because users are logged in for long periods with this type of application. Although each user consumes a small amount of application memory, there can be many concurrent users so the application memory consumption can become a bottleneck. For example, there were about 55,860 users active at peak capacity (931 converged operations per second * 60 seconds). The application memory that is used is part of the *Java heap.*

The used Java heap per node is measured by summing the steady state Java heap memory used by each application server.[5] As shown, the steady state memory consumption is 2,127 Mbytes total or about 266 Mbytes per application server. Since each application server was allocated a Java heap of 1,500 Mbytes this leaves roughly 83% of the application memory available. This memory measurement technique can also be used if the SIP or HTTP application is split across more than one tier.

[4] The term Converged Operations per Hour is based on prior terminology. *Converged* is the usual industry term for web applications that use both SIP and HTTP. Telephony capacity is usually measured as Busy Hour Call Attempts. The new term joins these two concepts together.

[5] The IBM JVM's generational garbage collector is used in these measurements. The garbage collector activity is logged in the file native_stderr.log when enabled. The steady state memory usage is found by searching for the last 'global' garbage collection (i.e., 'gc type="global"') at peak capacity. Then the average used memory is calculated as the tenured 'totalbytes' value less the tenured 'freebytes' value. Other JVM vendors will record similar information.

An important quality factor for SIP is the round trip delay between issuing a SIP request and receiving the acknowledgement. If the acknowledgement is not received in a timely fashion, then the client retransmits the message. This window of time is relatively small compared with HTTP time-outs. For example, the initial message time-out value for a VoIP call is 500 milliseconds while the typical HTTP time-out value is 30 seconds: a factor of 60. Although the WebSphere Application Server detects and discards retransmitted messages, the retransmissions use some processing power and also waste network resources, so minimizing retransmissions is beneficial. For this reason, the application round trip time is a useful quality measure because it provides an expectation of how close the system is to the retransmission threshold.

For the web voice mail benchmark, round trip SIP latency is measured as the interval from the voice mail server sending a NOTIFY to receiving the 200 OK acknowledgement. The 200 OK is generated by the application so measurement includes application latency; some other responses (e.g., 100 TRYING) do not reach the application but are returned by lower level protocol processing. The “95th Percentile NOTIFY to 200 OK Latency” measurement in Figure 3 shows that 95% of round trip times are less than 60 milliseconds, 440 milliseconds below the retransmission threshold.

4 System configuration and performance of highly available cluster

Application capacity can be increased by horizontally scaling the environment with additional nodes to form a cluster. A cluster can also be made Highly Available (HA), so that a user’s session will continue in the event of single failure in the cluster. Figure 4 is an example of a small HA cluster.[6] There are two nodes which each have eight application servers, just like the stand-alone example of Section 3. Each application server on one node is paired with an application server on the other node, exchanging session and protocol data so that the user session can survive an application server failure.

In comparison to Figure 2, the cluster configuration of Figure 4 has the additional WebSphere Proxy Server. The WebSphere Proxy Server includes the following main features:

- Application level session failover and load balancing of SIP and HTTP protocol requests;
- Manages the work offered to the application servers so that the application servers are not overloaded (e.g., a denial of service attack);
- Maintains user session affinity information.

The WebSphere Proxy Server acts as both a load balancer and message router. HTTP requests are first sent to the WebSphere Proxy Server which then forwards them to a WebSphere Application Server instance. The HTTP response flows back through the proxy server too. The SIP messages to, and from, the WebSphere Application Servers must traverse the WebSphere Proxy Server to enable the load balancing, overload control, and high availability features. This

[6] To avoid a single point of failure in this example, a second WebSphere Proxy Server is needed. From a pure performance perspective, a single proxy server handles the workload so adding an additional proxy server does not impact the performance results. So, to keep things simple a single WebSphere Proxy Server was used.

enables the WebSphere Proxy Server to act as a centralized message monitoring point, for diagnostics. Additional Proxy Servers can be added to the cluster to avoid a single point of failure.

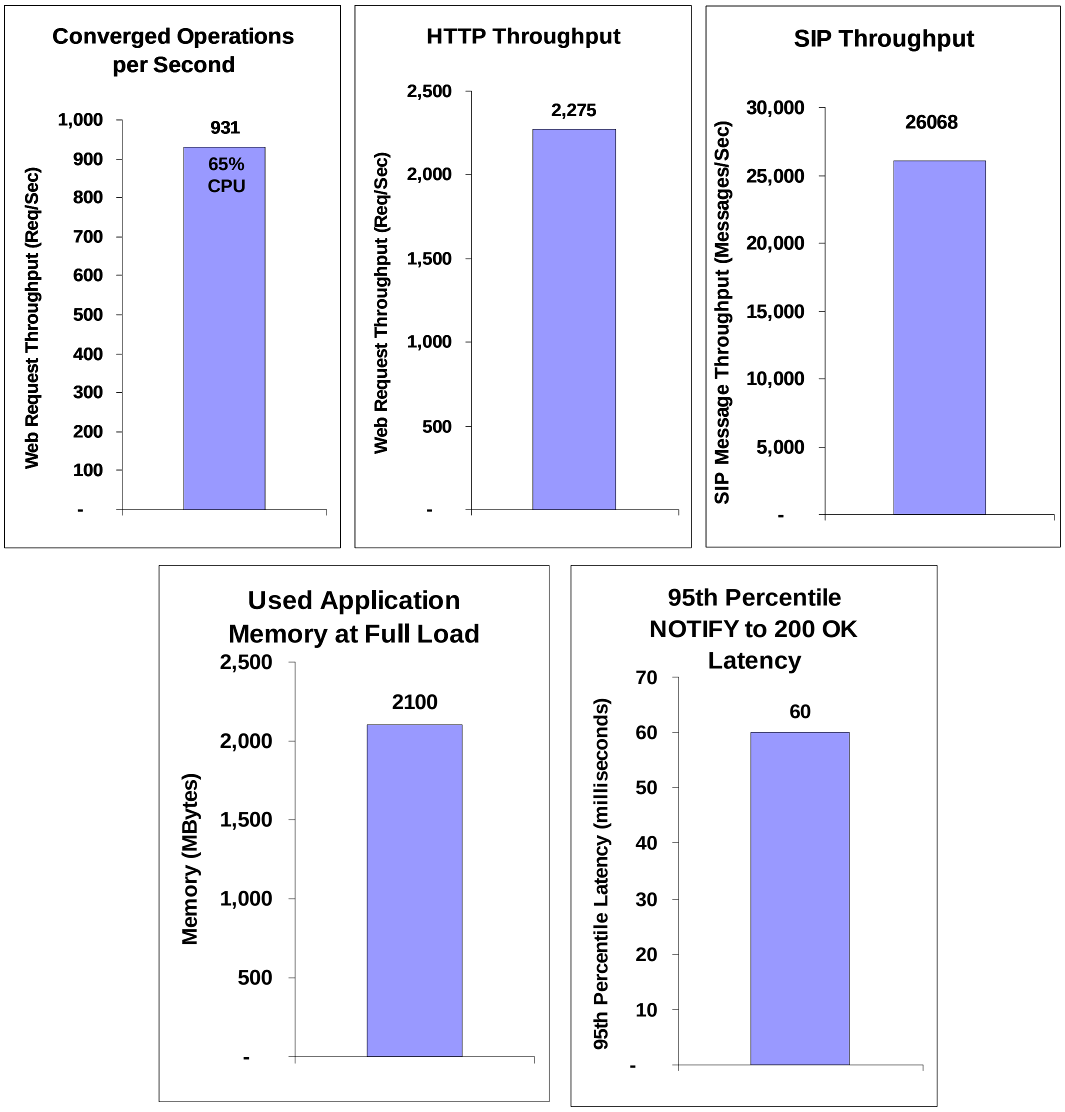


Figure 3: Performance of the Web Voice Mail Application on a Stand Alone Node

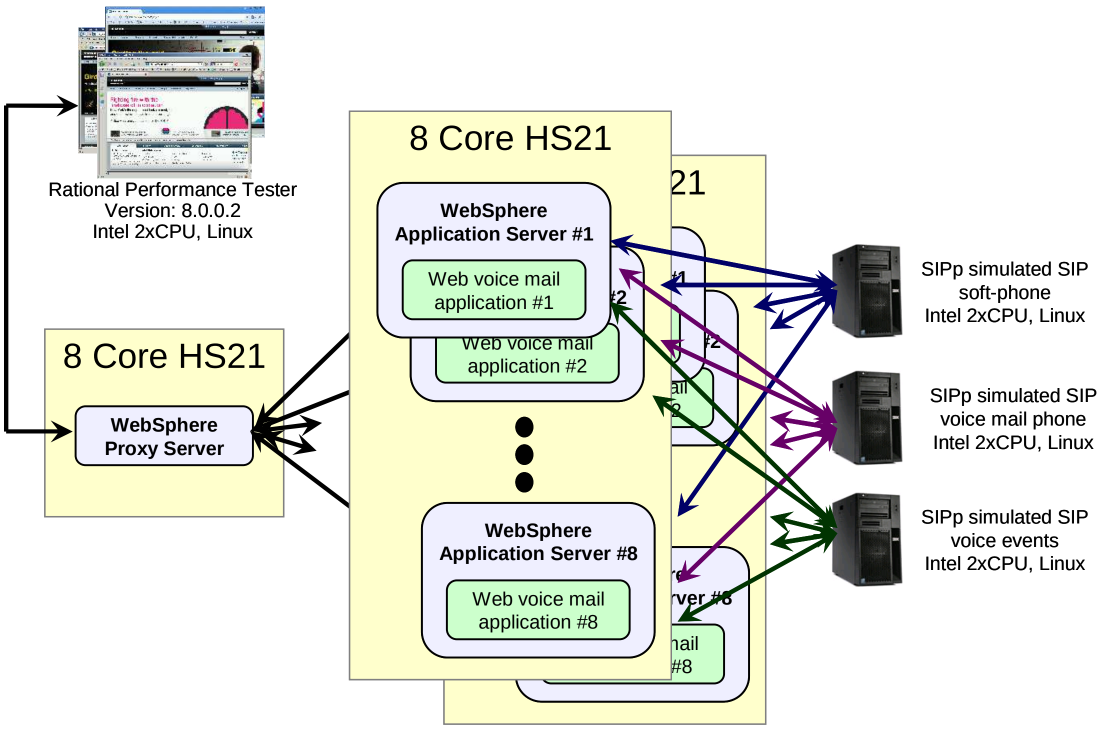

Rational Performance Tester
Version: 8.0.0.2
Intel 2xCPU, Linux
8 Core HS21
WebSphere Proxy Server
8 Core HS21
21
WebSphere Application Server #1
Web voice mail application #1
Web voice mail application #2
WebSphere Application Server #8
Web voice mail application #8
SIPp simulated SIP soft-phone
Intel 2xCPU, Linux
SIPp simulated SIP voice mail phone
Intel 2xCPU, Linux
SIPp simulated SIP voice events
Intel 2xCPU, Linux

Figure 4: A Small, Highly Available Cluster

The high availability benchmark configuration is shown in Figure 4 and expands the stand alone configuration of Figure 2. There are two application server nodes with eight WebSphere application servers configured per node. The application server node is an IBM BladeCenter® HS21 xSeries blade server with a two 3.16GHz Intel® quad core X5460 CPUs with 16 GBytes of RAM, running Red Hat Enterprise Linux release 5.2. The proxy server hardware node is an IBM BladeCenter® HS21 xSeries blade server with a two 3.33GHz Intel® quad core X5470 CPUs with 16 GBytes of RAM, running Red Hat Enterprise Linux release 5.2. The load generation configuration was unchanged from Figure 2.

The performance results of Figure 5 are shown per node so the entire capacity of the cluster would be twice these values. Each WebSphere Application Server node processed 349 converged operations per second in a High Availability configuration. This breaks down to an HTTP request rate of 873 HTTP requests per second (roughly 349 converged operations per second * 2.5 HTTP requests per converged operation), while the SIP request rate is 9,772 SIP messages per second (349 converged operations per second * 28 SIP messages per converged operation). This combined throughput was achieved with an average 72% CPU utilization for each WebSphere Application Server nodes. This is equivalent to 2,706,092 Busy Hour Call Attempts (assuming each call is 13 SIP messages) while also processing 3,142,800 HTTP requests per hour, on a node running the application in an HA configuration. This capacity is lower than the stand alone configuration of Section 3 due to HA using additional resources for the data replication to maintain the user and SIP protocol session state.

The memory consumption per node is increased from the non-HA application server because backup session information is used to recover the user sessions in the event of an application server fault. As shown, the steady state memory consumption per node is 1100 Mbytes total or about 136 Mbytes per application server. Since each application server was allocated a Java heap of 1,500 Mbytes, this leaves roughly 90% of the application memory available.

For the clustered configuration, the round trip SIP latency is still measured as the interval from the voice mail server sending a NOTIFY and receiving the 200 OK acknowledgement. Figure 5 shows that 95% of round trip times are less than 30 milliseconds, 470 milliseconds below the retransmission threshold.

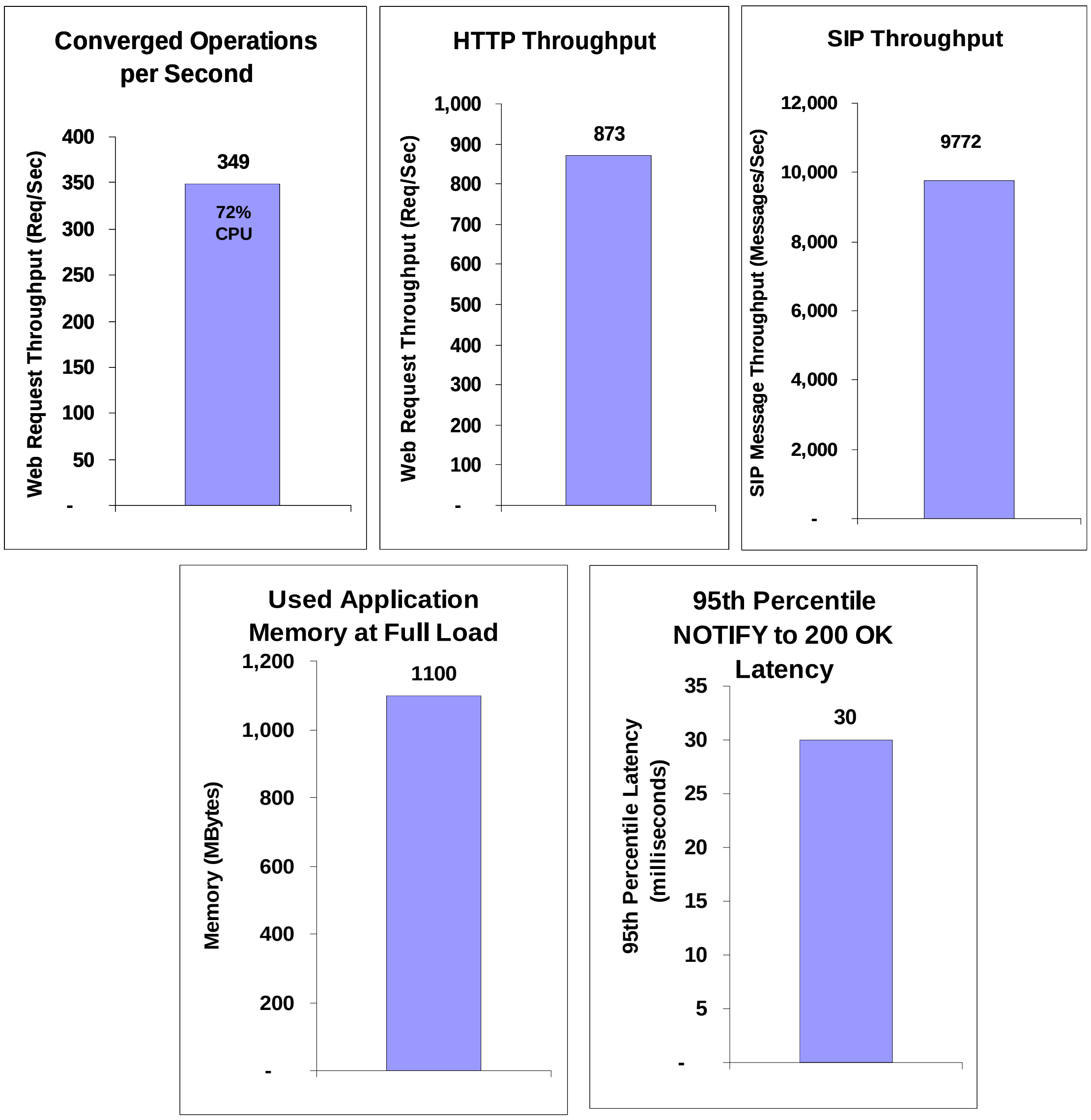


Figure 5: WebSphere Application Server Node Performance in a High Availability Cluster

5 HTTP and SIP Performance Differences

HTTP is a well known technology, so it is useful to contrast it with SIP to understand the similarities and differences around performance. This is done first for a pure web server and then for an application server.

HTTP uses the TCP protocol just like other common higher-level protocols (e.g., Telnet, FTP). From a pure HTTP web server perspective, the key HTTP performance measures are the peak web request rate along with its CPU utilization. Most of the tuning mechanisms that are available for a web server are TCP tuning values. For example, a TCP connection to a client looks like a file handle to the operating system and there is usually an operating system limit to the number of file handles that can be open at any time. This can be adjusted for each operating system but there is usually a limit of 64 thousand with a practical limit lower than that. An additional tuning factor is the number of threads to service request along with the associated threading mode.

A quality factor that is important to users of a web server is the time it takes for the web page to be rendered so that they can proceed with what they want to do. The response time measure is important because if it is too large then the user will abandon the web page and go on another web site, perhaps resulting in a lost sale or bad reputation for the original web site. This response time measure can involve many HTTP round trip message exchanges to the web server because a web page can have many embedded elements (e.g., images, style sheet, other referenced web pages in frames, etc.), as well as have internal scripts perform some operation which takes additional time. These round trip times may also involve multiple requests to backend databases so the latency involved with the database can impact the response time. Web Server response time is primarily a function of how the web page is retrieved which is well understood and not covered here.

An application server provides more services than an HTTP web server so it has additional tuning mechanisms. In general, the maximum throughput rate and the associated CPU utilization are the key metrics to optimize for. Some application server related tuning factors are: the number of threads in thread pools, the number of connections, connection pooling, JDBC data source statement cache size, and other caching mechanisms. For a good overview of tuning a JEE application server see the article [JeeTuning].

SIP and HTTP are similar because they use text requests and responses, but they differ in several ways that impact performance:

- SIP applications commonly use UDP as the primary protocol instead of TCP, so out-of-order message reception is tolerated by SIP. Applications may also have to consider out of order message reception;
- Some SIP requests need to explicitly receive an acknowledgement within a time limit, after which the request is retransmitted. UDP retransmissions consume network and CPU resources. TCP handles retransmission implicitly;
- SIP message transactions and sub-transactions may be stateful or stateless. Additional, long-lived memory is required to manage the stateful transactions for each user. HTTP requests are stateless;
- SIP messages are usually very small so they typically fit into a single MTU packet. HTTP responses frequently exceed an MTU packet;
- SIP interactions may be synchronous (like HTTP) but may also be asynchronous. SIP has a peer to peer architecture, rather than the HTTP client-server approach.

- More than one SIP response may be generated in response to a single SIP request. An application that uses SIP has the freedom to send out several requests for a given input, or wait for several inputs before a response. HTTP has a simple reply-to-request model; and
- SIP is a bursty protocol where messages clump together in the network so messages are usually sent and received in groups.

Due to the preceding factors, adding the SIP protocol to an HTTP environment introduces additional performance considerations and tuning factors. SIP communication over UDP provides performance advantages when compared with TCP. UDP avoids the TCP slow start and the overhead of the communication channel set-up and tear down. This advantage comes with some trade-offs, such as most operating systems are tuned for TCP communications and need to be retuned to support high performance UDP communication. Also, the SIP protocol needs to manage when UDP message retransmission occurs.

The asynchronous nature of SIP requires that the SIP protocol and application deal with retransmitted messages or messages that are received in a different order than sent. Also, if a SIP messages exceeds the MTU of a UDP packet, the SIP protocol does allow for TCP to be used to send the larger payload. However, mixing TCP and UDP messages can make tuning more complex. A recommended alternative is to leverage the converged application's HTTP protocol to transfer large data payloads. In this approach, the SIP message has an URI that the application uses to retrieve the large data payload. This allows SIP to use UDP consistently, as well as employing the HTTP protocol to serve up the large objects.

The asynchronous nature of SIP highlights strength of the WebSphere Application Server, which is its integrated, converged container (see Figure 6). A converged JEE application server needs to process HTTP and SIP messages, not only for the same application, but for the same user. This means that care must be taken to accommodate concurrent processing of an HTTP message, and one (or more) SIP messages.

The WebSphere Application Server helps the developer develop for this asynchronous environment. The container manages the session data independently for each user and serializes the access per user. This serialization allows user session data to be accessed from a SIP thread or HTTP thread without requiring special API's, or language constructs. This integrated, converged container makes it easier for the developer to program at the application level because there are fewer things to worry about (e.g., synchronizing access because of multiple SIP threads and HTTP thread needing to be processed).

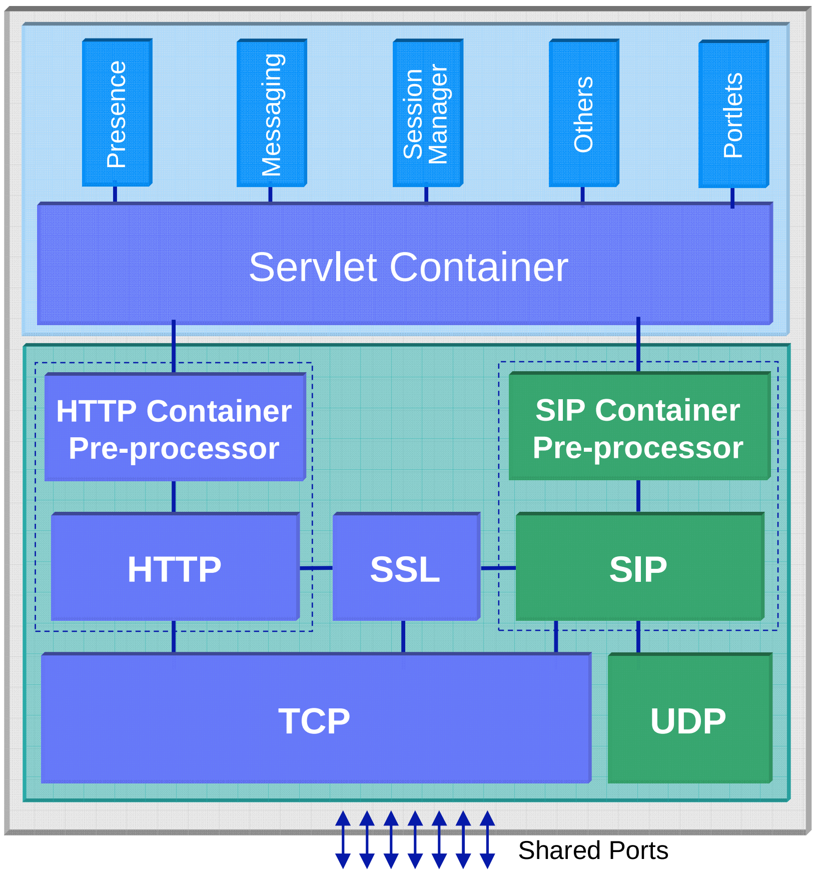


Figure 6: WebSphere Application Server Converged Container

6 Tuning Goals for SIP and HTTP

There are many books or articles that describe what goals should guide the tuning process of HTTP servers. As the previous section mentioned, combining SIP and HTTP changes the performance, requiring some adjustment to the tuning goals. These goals are elaborated here because they are used to guide the tuning process in the following sections.

Goal: Tune to avoid SIP UDP message retransmission by smoothing out the SIP traffic processing where possible. If the acknowledgement to a SIP request is not received in a timely fashion, then the client retransmits the message. This window of time is not large compared with HTTP time-outs. If SIP messages are retransmitted *en masse*, then a network storm can occur which can result in five times the expected network traffic. For example, if the normal SIP message rate is 3,000 SIP messages per second, then if the rate remains the same and acknowledgements are not received in time, the peak network traffic could rise to 15,000 SIP messages per second. [7] Although the WebSphere Application Server detects and ignores retransmitted messages, the retransmissions use some processing power and also waste network

[7] Although not discussed, the WebSphere Application Server has overload control mechanisms to protect it in this situation.

resources, so minimizing retransmissions is beneficial. For this reason, tuning to minimize UDP retransmissions is key goal. It also leads to other goals.

Goal: Engineer the system for 80% CPU utilization. CPU tuning for a SIP and HTTP application is like any other web application. There are obvious things to do (e.g., minimize object synchronization time, provision thread pools appropriately, avoid excessive logging). The large HTTP request timeout facilitates high CPU utilization approaching 100% by queuing HTTP requests because each request is usually serviced within the time-out window. But SIP's short retransmission timer means that request queuing latency needs to be managed. SIP's small message time-out does elapse if a large number of SIP requests are queued. So, a CPU utilization target of 100% is not appropriate. Instead, a target of 80% CPU utilization is recommended for the maximum load which allows for bursts of activity to be accommodated without violating the SIP time-out limits.

Goal: Minimize application stall time so SIP message acknowledgements are sent quickly. The Java Virtual Machine stalls the application processing when it performs a garbage collection to reclaim unused memory. SIP tuning should minimize this impact using two approaches: (i) minimize the number of garbage collection events (discussed in the next goal) and (ii) minimize the amount of time each garbage collection event takes. An infrequent activity that occurs during a garbage collection event is a *compaction* which relocates objects in memory to maximize contiguous free memory by shrinking the unused memory gap between the objects. Avoiding garbage collection compactions belongs to the second category because a compaction can take several seconds.

Application design can help to avoid compactions. First, pre-allocate at start-up any large objects. Secondly, when possible, pre-allocate large collections with their expected maximum size. When a collection grows in size it needs a chunk of contiguous memory which is not a problem at start-up since the heap is not fragmented. However, after the application runs for a while, a very large contiguous chunk of memory may be hard to find because the heap could become fragmented through typical use. Lastly, the IBM JVM provides a way to avoid compactions when allocating objects larger than 64 Kbytes by reserving an area of the heap called the Large Object Area (LOA). The LOA is reserved at start up and it is discussed further in the [JvmDiagnostic] guide.

Goal: Minimize memory usage but assume a lot of memory will be used. Memory consumption is an important consideration because these application types have user sessions typically active for long periods. Although each user consumes a small amount of application memory, there can be many concurrent users so the total application memory consumption can become large. In addition to the user data, long-lived memory for the SIP protocol state of each user is also retained. This can be very large since the number of user application sessions is the average request rate multiplied by the average lifetime of the sessions.

For example, if users' sessions live for an average length of three minutes and the request rate to an application server is 200 requests per second, then an application server will have 36,000 live application sessions (200x3x60). If the application behaves as a SIP back-to-back user agent (B2BUA) then there will be at least two SIP sessions per user, for a total of 72,000 SIP sessions. If each application session uses 25 Kbytes, then this means that 900 Mbytes of Java heap memory will be in constant use. If the call hold time increases by an additional minute, then the live heap usage increases to 1.2 GBytes because the number of sessions increases by 33%. This is a large amount of active memory which will impact the time it takes to perform a garbage collection event.

Some application design decisions that minimize memory consumption are: keep the data stored in the user session to a minimum; use soft-references or weak-references in object caches so that the cache memory becomes available if there is a lot of memory pressure; invalidate a session as soon as possible by using the appropriate method; and adjust the default time-out values to your application's characteristics.

Goal: Use a 32-bit JVM if the SIP messages have QoS requirements, otherwise use a 64 bit JVM. Obviously, there must be enough memory available for the application during the peak load. If the application memory of a single server does not support the peak load, then there are two alternatives: (i) use a 64-bit JVM or (ii) vertically scale the application by running multiple 32-bit JVMs (even if the operating system is 64 bit).

Although a 64 bit JVM can address more memory this extra address space comes with some additional costs. For example, each 64 bit JVM object reference requires more memory than the 32 bit JVM object reference.[8] More importantly, using a very large heap with a 64 bit JVM has the side effect of increasing the garbage collection times which may stall the application for a longer time and make retransmissions more probable. A 64 bit JVM can be used if the SIP message exchanges do not have QoS requirements for receiving acknowledgements. For example, the SUBSCRIBE and NOTIFY SIP messages do not have retransmission time-out values so applications which use only these messages do not need to be tuned for message latency. The IBM JVM has an option which reduces the impact of object reference size by using a feature called compressed references which is enabled by the "`-Xcompressedrefs`" option.

Running multiple JVM's per WebSphere node can add more capacity when memory usage is a bottleneck because it effectively adds more Java application memory, although this application memory is spread across the several WebSphere application servers. If the sessions have a long call hold time (e.g., three minutes) or live for long periods then this is a good strategy to use.

Vertical scaling can result in significant additional capacity, even though a portion of the additional memory is used for each WebSphere application server run-time. For example, suppose there are three JVMs executing on the same node where each application server has 1.5 GBytes of free memory and the run-time memory footprint of the application server is 150 Mbytes. Then there is roughly 4 GBytes of application memory available, spread across three JVMs. This additional memory translates into additional session capacity and, usually, less garbage collection activity.

Goal: Minimize queuing in the network communication path by servicing requests quickly. This can be accomplished in two ways. The first is to reduce the network processing delay for which operating system tuning is the primary tool. The second is to introduce multiple queues that are serviced in parallel to reduce the delay. For example, an operating system socket can be a bottleneck for an extremely high SIP message rate because the socket is not serviced quickly enough, causing requests to queue. Multiple sockets in an application server for sending and receiving can alleviate this. Vertical scaling strongly supports this goal because it provides

[8] The IBM JVM has an option that reduces the size of the object references for the 64 bit JVM. See [64BitJVM] for more information.

concurrent socket communication through separate ports with different thread pools. Both of these approaches use multiple sockets to reduce thread contention for receiving and transmitting messages, as well as reducing lock contention for concurrent access to the socket.

Goal: Provide adequate communication buffers so that message queues do not overflow. Due to the bursts in message reception, adequate communication buffering is needed. With inadequate buffering, messages may be received and dropped, resulting in a message retransmission. This is especially important when it is recognized that there is the potential for a positive feedback effect that compounds: dropping a packet results in a retransmission which use more communication buffers, resulting in more dropped packets because fewer communication buffers are available, so there are more messages retransmitted, etc. All the various network layers (i.e., network card, operating system buffers, etc.) must be tuned to ensure there is adequate buffer capacity. Vertical scaling strongly supports this goal because each application server instance adds additional communication buffers at the application level (e.g., socket) so that, overall, there is sufficient communication buffer capacity.

To conclude this section, a summary of the tuning goals are:

- Tune to avoid SIP UDP message retransmission by smoothing out the SIP traffic processing.
- Engineer the system for at most 80% CPU utilization under the heaviest expected workload.
- Minimize application stall time so SIP message acknowledgements are sent quickly.
- Minimize memory usage but assume a lot or memory will be used.
- Use a 32-bit JVM. A 64 bit JVM can be used when SIP message exchanges do not have QoS requirements
- Minimize queuing in the network communication path.
- Provide adequate communication buffers so message queues do not overflow.

The tuning processes follow from these goals.

7 Tuning a stand-alone WebSphere application server

The tuning for a single application server is discussed in this section, with the subsequent section describing additional cluster tuning. This tuning is not necessary for a functionally working system but it can provide significant performance gains. The tuning is also applicable to configurations that are larger than a single application server, such as a WebSphere Network Deployment cluster.

The tuning is described in the following sections, moves from the lowest to highest level in the software stack. The operating system is the lowest level in the software stack and it is discussed first. Then the JVM tuning is presented, which is the largest section. Then the application server tuning is discussed. Finally, some application level tuning guidelines are presented.

7.1 Operating system tuning

As mentioned, most operating system's network parameters are tuned for TCP so tuning for UDP performance is needed. Network tuning for the WebSphere Application Server has been extensively covered in [CMG] and [Info1] so it is not reviewed here. It is reproduced in Table 3 for completeness.

There are a several other operating system tuning factors that should be considered. For Linux operating systems ensure the "Name Service Caching Daemon" (nscd) is running to avoid slow hostname resolution. The network capacity can be increased by using multiple Network Interface Cards (NIC). A technique to achieve this is called *link aggregation* (also known as *trunking* or *bonding*) which can bind one IP address to multiple, active NIC cards. Link aggregation was not used in the benchmark measurements but is mentioned for completeness. If possible, IPv6 should be disabled. This is accomplished by adding the following lines to the file '`/etc/modprobe.conf`': "`alias net-pf-10 off`" and "`alias ipv6 off`". Lastly, ensure that an adequate number of sockets can be opened by increasing the number of operating system file handles. The '`ulimit`' command is used to do this.

7.2 JVM tuning

The largest group of tuning factors is associated with the JVM garbage collection options. Garbage collection (GC) is performed by the JVM to reclaim application memory for reuse. During the GC process the application is stalled so a tuning goal is to minimize this delay. The questions to answer are:

- Which garbage collection algorithm to select?
- What should the size of the Java heap be?
- How should the Java heap be partitioned?
- What settings can be used to avoid compactions?
- Are there any additional JVM settings that enhance performance?

The options are covered here in increasing priority order.

7.2.1 Selecting the garbage collection algorithm

Selecting the appropriate GC algorithm is the first step because this choice affects the subsequent tuning parameters. The recommended GC algorithm is the *generational garbage collector* (called *gencon*). Gencon is a compromise of minimizing application latency and maximizing throughput, as well as providing several useful tuning factors. A little background about the gencon garbage collector helps to understand the tuning. For a detailed discussion, the reader is referred to [JavaDiagnostic]. These articles are also helpful [GcGencon] and [GcPerf]. The [JeeTuning] article has some examples with freely available garbage collection analysis tools.

A unique feature of gencon is that the heap is divided into two areas: one area of the heap for newly created objects (the new object or *nursery* space) and another area for long lived objects (the old object or *tenure* space). The garbage collection activity operates in two modes: focusing solely on a nursery garbage collection, or performing both a nursery and tenure space collection. The nursery uses up its free space more frequently than the tenure space because objects are first allocated there. When the nursery space is used up then the application is suspended and a *scavenge* GC is performed that recovers memory in the nursery space. During the scavenge GC event, objects are promoted (moved) from the nursery space to the tenure space if they have lived long enough in the nursery space. The result is that the tenure space stores the long lived objects. If, during a nursery garbage collection event, there is no more space to move objects from the nursery space to the tenure space, a *global* garbage collection occurs which also reclaims memory in the tenure space.

The JVM option that selects the generational GC algorithm is '`-Xgcpolicy:gencon`'.

7.2.2 Determining the size of the Java heap

The size of the Java heap is selected by balancing the three factors: the amount of application memory in constant use, the amount of RAM memory available, and the length of the garbage collection events. To select an optimum Java heap size some testing is needed, which is covered in the next sub-section. In general, a Java heap of 1.2 GBytes is good starting point because it provides good performance for a typical SIP application (called a Back-to-Back User Agent application) with an average call hold time of three minutes. If the average call hold time or session lifetime is less than three minutes, then the heap size could be decreased. If the average call hold time is longer than the heap size could be increased. If each user session stores more than 2 Kbytes of user data in the session then the heap size will also need to be increased. A Java heap size of 2 GBytes can be used (assuming a 32 bit JVM). When sizing the heap, it must be ensured that the full heap resides in RAM and is not swapped to disk by the virtual memory system.

For server applications, it is a good practice to set the minimum and maximum heap values to the same value because this avoids a delay when the Java heap size decreases. Using different minimum and maximum heap sizes allows an application's use of memory to grow or shrink. However, reducing the heap size behaves like a compaction event, where the application stalls while objects are moved in the new heap to decrease the heap size, etc. It is recommended that the minimum and maximum heap sizes be fixed at the same value for the WebSphere application server to avoid this.

The JVM option that sets the minimum heap size is '`-Xms<size>`'. The JVM option that sets the maximum heap size is '`-Xmx<size>`'. The following options would set the minimum and maximum Java heap size to 1200 Mbytes: '`-Xms12000m –Xmx1200m`'.

7.2.3 Partitioning the Java heap

Careful selection of the relative sizes of the nursery and tenure space areas can minimize the application delay in two ways. An obvious approach is to minimize the time each GC event by tailoring the size of each area, ensuring it is not too large. Guidelines for how large the areas can really be are based on the amount of time the garbage collection events take. The recommendation is to select the nursery and tenure space size so the time for one nursery scavenge event and one global event take less than 500 milliseconds most of the time. An initial target for the scavenge garbage collection is 150 milliseconds. A target for the global garbage collection is 350 milliseconds.

The other approach ensures that several GC events do not occur in the same second which limits application processing time, even though a single GC event is not very large. This is illustrated by Figure 7 where three GC events occur (two nursery and one global) within the same second.

What is significant is that out of that entire second, only about 150 milliseconds is available for the application and this is too little time to process all the queued up requests, leading to retransmissions. Each GC event is less than 500 milliseconds so a cursory review may not notice that the closely spaced GC events do not allow enough application processing time.

These closely spaced or cascading garbage collection events occur when: (i) a nursery scavenge GC event begins and received messages are stored internally by the operating system or socket; (ii) the queued up messages begin to be processed but they use up the available nursery space; (iii) then a global GC event is triggered during this nursery GC event because there isn't enough tenure space to promote objects; (iv) during the global GC event, further messages are queued; (v) when the global GC event finishes, another nursery GC event occurs during the processing of the queued work. This type of activity can happen when there is a very high inbound message rate so that the nursery space is quickly used in processing the queued messages.

These two approaches to sizing the GC areas has a range of trade-offs between making individual events short but having many events, and the other extreme of only having large GC events.

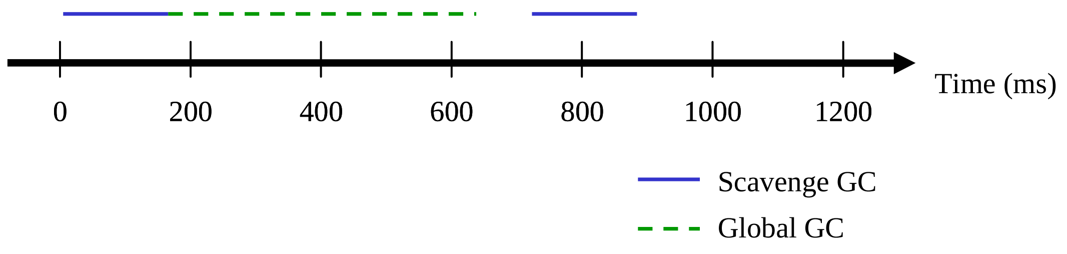


Figure 7: Closely spaced garbage collection cycles

Setting the nursery and tenure size is done with one option: '`-Xmn<size>`' which sets the size of the nursery. Values for the nursery size generally range from '100m' (100 Mbytes) to '250m' (250 Mbytes). The size of the tenure space is calculated as the size of the Java heap, less the size of the nursery. For example, the following set of options : '`-Xms12000m –Xmx1200m –Xmn150m`' establishes a Java heap size of 1200 Mbytes, a nursery size of 150 Mbytes, with a corresponding tenure space size of 1050 Mbytes (1200 Mbytes – 150 Mbytes).

Tuning the heap size does require running the system under a heavy test load, monitoring the garbage collection time, and then adjusting the nursery or tenure space. The overall steps are:

1. Examine the nursery garbage collection time for the nursery garbage collections. If the time is too large then decrease the nursery size.

2. Count the number of nursery garbage collections that occur within a one second period. If there is more than one nursery garbage collection then increasing the nursery size may be advantageous.

3. Measure the global collection time. If the time is too large then decrease the tenure space size by reducing the Java heap.

4. Check to see if there is a large number of cascading GC events. If there are, then increase the nursery size.[9]

5. Adjust the JVM parameters, restart the application server, and rerun the workload.

Each of these steps is described using the example text of Figure 8.

```
<af type="nursery" id="1078" timestamp="Aug 28 00:47:51 2009" intervalms="557.296">
    … text deleted …
  <time totalms="80.338" />
</af>
<con event="collection" id="9" timestamp="Aug 28 00:47:52 2009" intervalms="63618.373">
    … text deleted …
  <time totalms="322.192" />
</con>
<af type="nursery" id="1079" timestamp="Aug 28 00:47:52 2009" intervalms="899.301">
    … text deleted …
  <time totalms="93.525" />
</af>
<af type="nursery" id="1080" timestamp="Aug 28 00:47:53 2009" intervalms="799.301">
    … text deleted …
  <time totalms="90.125" />
</af>
```

Figure 8: Garbage collection text examples

There are four garbage collection events in Figure 8, with some text removed to highlight the parts of interest. There are three nursery events which are identified by the ‘type="nursery" ’ label and one global event (the ‘event="collection" ’). Each event records the amount of time that the event took in the last "<time totalms=" tag, which is the time (milliseconds) the application was stalled. To determine the time between garbage collection events of different types, the "timestamp" values need to be subtracted; the "intervalms" events cannot be used because they are the time between events of the same *type*.

Looking at the values of these events, the nursery and tenure size are good because the scavenge GC times are below 100 milliseconds and the global event is below 350 milliseconds. By examining the timestamp values it can be seen that the global GC event and subsequent nursery GC event both occur in the same second but their total time would only stall the application for 415 milliseconds which leaves 585 milliseconds for the application. Additionally, the third nursery GC event, with ID of 1080, occurs in the next second so it does not need to be included. In summary, the events in Figure 8 suggest that the application is well tuned.

7.2.4 JVM settings to avoid compactions

A JVM compaction takes a large amount of time. The JVM tuning described here is to avoid compactions where possible. The key JVM options, with descriptions, are shown in Table 1. The first two options are strongly recommended, while the last option can avoid compactions when the system experiences a sudden spike-like increase in the workload.

[9] The main cause is the nursery-global-nursery sequence. This can be avoided by increasing the nursery size which greatly reduces the probability of that third nursery GC event. There is a slight increase in the nursery GC event duration but it is not a linear relationship since the nursery GC time is proportional to the number of live objects and not the nursery size.

JVM Option	Required (Y/N)	Description
`-Xloaminimum<value>`	Y	Set aside a portion of the Java heap so that new, large objects will be stored here to avoid a compaction. A typical value would be 0.01 which allocates 1% of the heap to the Large Object Area (LOA). This would be entered as '`-Xloaminimum0.01`'.
`-Xgc:scvNoAdaptiveTenure,scvTenureAge=1,stdGlobalCompactToSatisfyAllocate`	Y	Nursery objects which survive one garbage collection cycle will be promoted to the tenure space on the next garbage collection cycle. This is based on the assumption that if an object survives for the time between a nursery GC (ranging from 1 to 10 seconds), then the object is likely to survive for the duration of the web activity or call. Aggressive compactions are disabled.
`-Xcompactexplicitgc`	N	Enable compaction on a System.gc(). In some benchmarks, it has been found that doing a compaction after start-up can improve performance by increasing data locality for common system operations.

Table 1: Application Server JVM Tuning Recommendations to Reduce Compaction Events

7.2.5 Additional JVM settings

There are some additional JVM options that are useful.

The option '`-verbosegc`' reports garbage collection event information to the file native_stderr.log. This information is extremely useful for analysis. The reader is referred to [JeeTuning] which provides examples of examining this data using some of IBM's tools.

The option '`-Xlp`' enables the JVM to use large memory pages for the heap which makes JVM memory access more efficient. This does require some operating system support and the reader is referred to [JvmDiagnostic] for details. Using this option can increase capacity by 1-3%.

A JVM network communication option for buffers is “-XX:MaxDirectMemorySize=<memory size in Bytes>” is used to specify the amount of RAM that is used to buffer messages. SIP can use a lot of buffers so it is recommended to increase the default value of 64 MBytes. The example “-XX:MaxDirectMemorySize=128000000” increases the buffer size to 128 MBytes.

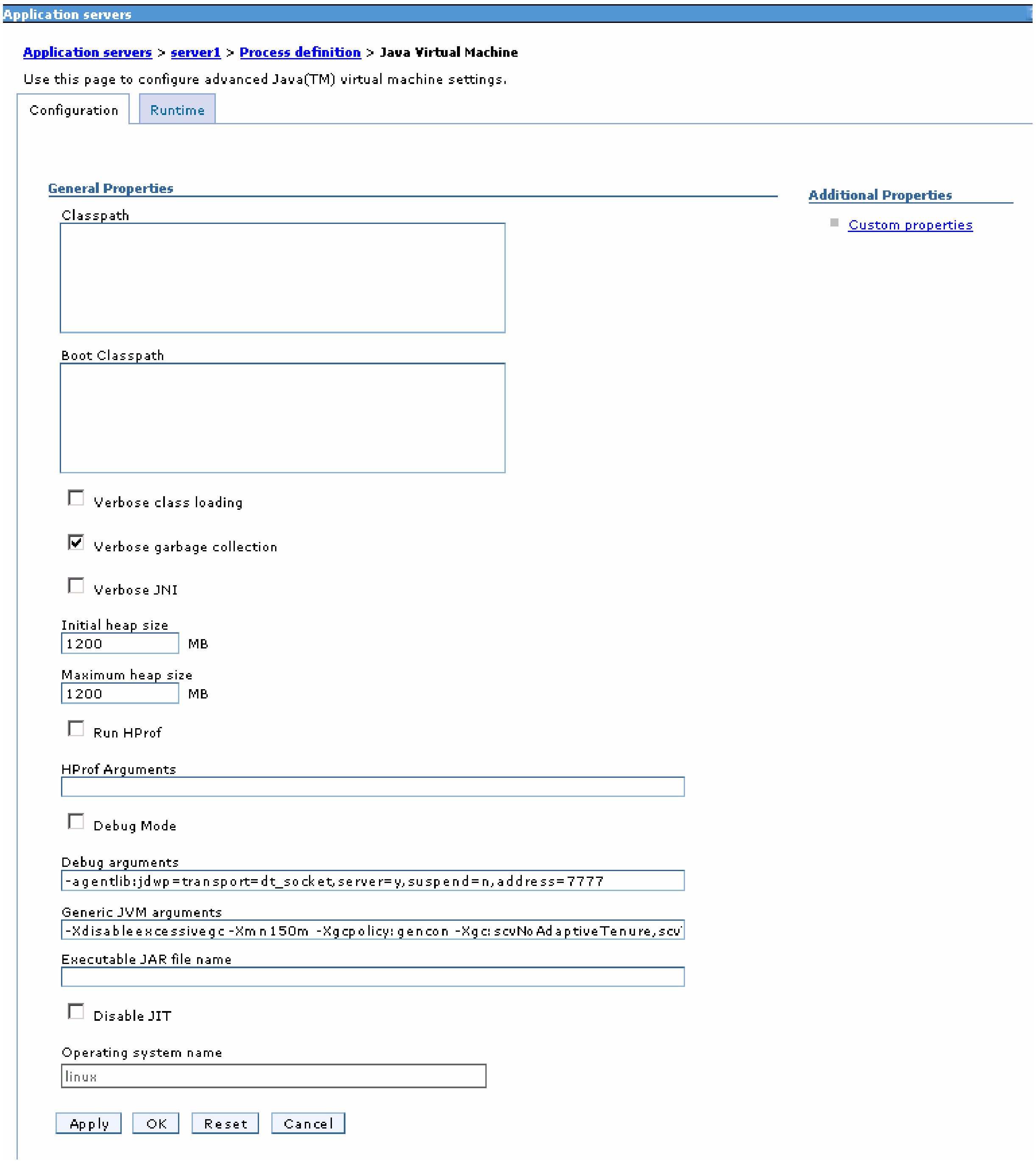
The last option ‘`-Xdisableexcessivegc`’ can prevent an out of memory exception being thrown if there is a lot of garbage collection activity. This is useful if the system is experiencing an extended spike in workload and there is a lot of GC activity. If this option is not used in this type of circumstance, it is possible that an out of memory situation is assumed by the JVM because much of the processing time is spent in garbage collection activity. Please see the [JvmDiagnostic] guide for further information.

7.2.6 Setting the JVM options

The WebSphere Application Server administration console is used to enter the JVM options. Figure 9 shows the console page, which is arrived at by the following path of menu/options: “Application servers” → <application server name> → “Java and Process Management” → “Process definition” → “Java Virtual Machine”. To enter the options do the following:

1. Check the “Verbose garbage collection” box if you want the garbage collection activity logged. This is the same as the ‘-verbosegc’ option;
2. Set the “Initial Heap Size” field to the chosen heap size. This is the same as the ‘`-Xms`’ option.
3. Set the “Maximum Heap Size” field to the chosen heap size. This is the same as the ‘`-Xmx`’ option.
4. Set the “Generic JVM arguments” field to the values “-Xgcpolicy:gencon –Xloaminimum0.01 –Xmn150m -Xgc:scvNoAdaptiveTenure,scvTenureAge=1,stdGlobalCompactToSatisfyAllocate” for a nursery size of 150 Mbytes.
5. Press the “OK” button at the bottom of the page.
6. Save the configuration changes.
7. Restart the application server for these changes to take effect

If additional options are needed, they can just be added to the string in step 4.



Application servers

Application servers > server1 > Process definition > **Java Virtual Machine**

Use this page to configure advanced Java(TM) virtual machine settings.

Configuration | Runtime

General Properties

Classpath:

Boot Classpath:

☐ Verbose class loading

☑ Verbose garbage collection

☐ Verbose JNI

Initial heap size: 1200 MB

Maximum heap size: 1200 MB

☐ Run HProf

HProf Arguments:

☐ Debug Mode

Debug arguments: -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777

Generic JVM arguments: -Xdisableexcessivegc -Xmn150m -Xgcpolicy:gencon -Xgc:scvNoAdaptiveTenure,scv

Executable JAR file name:

☐ Disable JIT

Operating system name: linux

Additional Properties

- Custom properties

Figure 9: WebSphere Console for Updating JVM Options

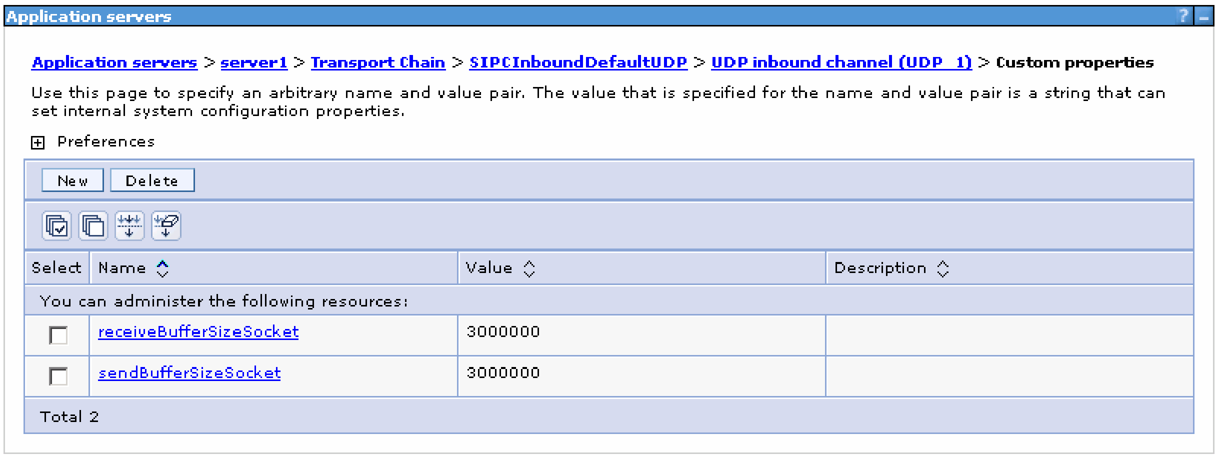
7.3 Application server tuning

The application server is the next layer in the software stack to tune. The tuning for a WebSphere stand-alone application server (i.e., not in a cluster) is straightforward. WebSphere application server does have mechanisms for managing an overload condition but they are not covered here.

A key WebSphere Application Server tuning parameter is the socket buffer size, for both sending and receiving messages. This is adjusted by adding custom properties to the SIP communication channel, as shown in Figure 10. This screen is arrived at by the following path of menu/options: “Application servers” → <application server name> → “Transport Chain” → “SipCInboundDefaultUDP” → “UDP inbound channel (UDP 1)” → “Custom properties”. The following property names and values can be entered:

Custom property	Value	Description
`receiveBufferSizeSocket`	`3000000`	Receive socket buffer size, in Bytes.
`sendBufferSizeSocket`	`3000000`	Sending socket buffer size, in Bytes.

These values may need to be adjusted for some applications but they have proven to be generally applicable in performance stress tests.


Application servers

Application servers > server1 > Transport Chain > SIPCInboundDefaultUDP > UDP inbound channel (UDP 1) > Custom properties

Use this page to specify an arbitrary name and value pair. The value that is specified for the name and value pair is a string that can set internal system configuration properties.

Preferences

New Delete

Select	Name	Value	Description
You can administer the following resources:			
☐	receiveBufferSizeSocket	3000000	
☐	sendBufferSizeSocket	3000000	
Total 2			

Figure 10: SIP Channel Tuning Values

7.4 Tuning the application

The application is the last layer in the stack to tune. A brief description of the implementation of the web voice mail SIP and HTTP application is given for those unfamiliar with JSR116 or JSR289.

There are three key parts to the application: the high level structure, the application state information of each user, and the application behavior in response to messages.

The high-level application structure derives from the JSR289 specification. It consists of two Servlets (one SIP and one HTTP) that interact through the common application session state of the user. The threads that run in each servlet are independent of each other. Any thread can update and store information about a user’s request, application, or SIP responses in the user’s application session state. The application information for each user is stored in the application session state. If high availability is enabled then the session state is replicated so that the application can continue even if a server fails.

After initialization, the behavior of the application follows a request-response model. When either an HTTP or SIP message is received, a corresponding method is called. A SIP message reception results in a JSR289 method call in the SIP servlet and the associated message object is used to navigate to the user's application session state information. Similarly with an HTTP request.[10] So, in the example application, each user's application session (SipApplicationSession) will have a single HttpSession, as well as a SipSession for the soft-phone, a SipSession for the voice mail server's VOIP call, and a SipSession for the notifications of subscribe/notify.

Although it is not a tuning mechanism, a general guideline is to introduce concurrency in the application only when it is absolutely warranted. In the example web voice mail application, the end of the call could be done in parallel and it would be very fast. However, this speed is not needed and it introduces additional complexity because there would need to be synchronization to prevent simultaneous access to the session state. Instead, the call is ended in a serial fashion: first close down the user phone call, then close down the voice mail server voice communications, and then the event messaging. This eliminates the need for synchronization as well as makes diagnosing problems easier since the application is easier to understand.

Where concurrency is advantageous, the Asynchronous Invocation API of the Communications Enabled Applications feature pack provides useful capabilities. The Asynchronous Invocation API simplifies how the developer considers sessions. For a single server, it is useful for serializing access to the same SIP application session when multiple threads are competing. In a clustered environment, this API can redirect work to a SIP session which avoids having a central session repository for all application servers and having sessions migrate between application servers. It allows work to be forwarded from one application server to another so that sessions do not need to be migrated between servers for more advanced interactions. For example, if a web-service request needs to interact with a SIP session on another application server, the Asynchronous Invocation API is used to forward the task to the remote SIP session.

A diagnostic tool that has proven useful is to instrument the application to print out the number of application sessions that are outstanding at a periodic rate. This information is useful for monitoring heap usage and correlating that with the number of live users.

8 Tuning a highly available cluster

As would be expected, an HA cluster has additional tuning related to the high availability mechanisms, as well as the WebSphere Proxy Server. The prior discussions about the stand alone application server tuning applies to the application servers in a cluster.

[10] This is done in the following manner. Each message that is received can navigate to an associated session object that corresponds to the message type: a SIP session (SipSession) or an HTTP session (HttpSession). Each of these session objects can then navigate to the user's application session (SipApplicationSession). Navigation can proceed in the other direction: from the user's application session to the SIP or HTTP session. For example, a SipSession can access the HttpSession by first getting a reference to the SipApplicationSession that is then used to reference the appropriate HttpSession. Each of these session objects has a unique identifier.

8.1 WebSphere proxy server tuning

The proxy server tuning is different than the application server because the proxy server has a different purpose. Unlike the WebSphere application server, the WebSphere proxy server has very little long lived memory because it does not need to maintain protocol information for each user. Rather, the proxy server simply passes messages along. This affects the proxy server tuning.

Proxy server goal: Minimize the duration of each garbage collection event. The proxy server introduces delay for each request that passes through it. The worst case delay occurs when a given message is delayed by both a WebSphere Proxy Server GC event and a WebSphere Application Server GC event. So, minimizing the additional proxy sever delay is done by adding a little delay to many requests rather than introduce a large delay for a small number of requests. This satisfies the objective to minimize the possibility of SIP retransmission. In summary, the GC tuning for the WebSphere Proxy Server is to have very short GC times, with the trade-off that GC events may be frequent.

This goal is achieved by the following configuration steps:

- Minimize the overall size of the Java heap to minimize the longest global GC event time;
- Make the nursery size much larger than the tenure space size because a nursery GC takes less time than a tenured GC. This works for the WebSphere Proxy Server because when a garbage collection event occurs most of the memory in the nursery space is no longer used so the scavenge operation is very fast;
- Keep objects in the nursery size for a long time to ensure that only really long lived objects are promoted to the tenure space;

To achieve this tuning, a different set of garbage collection tuning options are used for the proxy server, as shown in Table 2.

JVM Option	Required (Y/N)	Description
-Xgcpolicy:gencon	Y	Selects the generational garbage collection policy.
-Xmo160m	Y	Sets the tenure size to 160 Mbytes so that tenure GC events are very short.
-Xms650m	N	Sets the minimum heap size to 650 Mbytes. The maximum nursery size is 490 Mbytes (=650 Mbytes – 160 Mbytes).
-Xmx650m	N	Sets the maximum heap size to 650 Mbytes. The maximum nursery size is 490 Mbytes (=650

JVM Option	Required (Y/N)	Description
		Mbytes – 160 Mbytes).
-Xgc:scvNoAdaptiveTenure,scvTenureAge=8,stdGlobalCompactToSatisfyAllocate	Y	Objects which live for at least eight nursery GC cycles are promoted. Aggressive compactions are disabled.

Table 2: WebSphere Proxy Server Garbage Collection Tuning

8.2 High availability WebSphere application server tuning

There is additional cluster tuning for the high availability feature. When discussing high availability performance, there are three system states that need to be considered independently: behavior prior to a failure, behavior during a failure recovery operation, and behavior after a failure recovery operation.

During normal operation data is replicated from each application server to its active backup application server so that the backup can take over the duties of a failed application server. This data replication has an impact on CPU, memory, and network capacity. The additional CPU cost is to package the replicated data, transmit it to the backup application server, and to unpack replicated data that it receives. The additional memory cost is to package the data to be replicated (transient objects); the additional communication buffers to transmit and receive the replicated data; and the storage of the replicated data. The replicated data is stored in an efficient binary structure. Network resources are used to send and receive the replicated data but there is usually enough network bandwidth so this is not a concern.

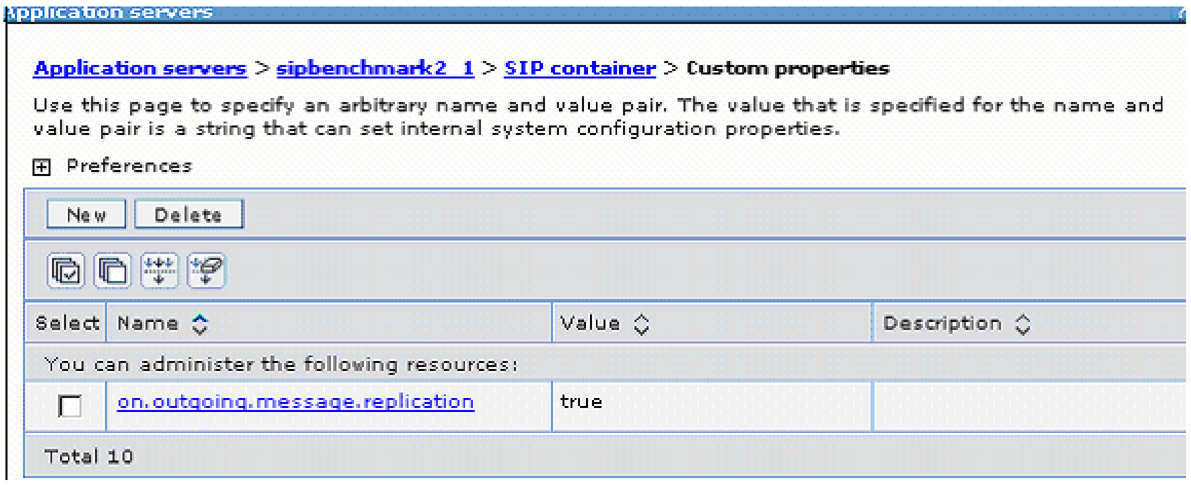
After a failed application server’s work has been recovered by its backup, the resource usage of the backup application server changes because data is no longer being replicated. However, this reduction in replication activity is offset by the new activity due to the sessions that have moved over to the backup application server. This new activity involves processing new messages for the failed over sessions, as well as any other work that is redirected to this backup that would have gone to the failed server. Although memory resources are no longer consumed for replication of data there is additional memory used by the backup session data which is now active, as well as the additional load that is directed to the backup application server.

When a failure is being recovered, the WebSphere Application Server takes care to not starve the application. This is done by lazily deserializing the failed server’s session information on a background thread. If a request is made for a session that has not yet been deserialized, that session is immediately deserialized and the application progresses. This approach amortizes the cost of the failure recovery over several seconds so that the application can continue to process the existing and new workload, with little impact.

The cost of the data replication is dependent upon the quality of service (QoS) that is requested. Replication QoS is configured via a SIP container custom property. There are three replication QoS options:

- Immediate replication: When a change is made to the session state, this change is immediately replicated on the backup application server. This has the highest resource usage. This is enabled by setting the property "`immediate.replication`" to "`true`".
- On outgoing message replicate: When a message is sent the data is replicated. This has intermediate resource use. This is enabled by setting the property to "`on.outgoing.message.replications`" to "`true`".
- End of service replication: All changes to application sessions are batched up and sent when the message processing ends. This has the least resource usage. This is enabled by setting the property to "`end.of.service.replication`" to "`true`".

The default is end of service replication. Figure 11 is an example of enabling replication to occur on each outgoing message.


Application servers

Application servers > sipbenchmark2_1 > SIP container > Custom properties

Use this page to specify an arbitrary name and value pair. The value that is specified for the name and value pair is a string that can set internal system configuration properties.

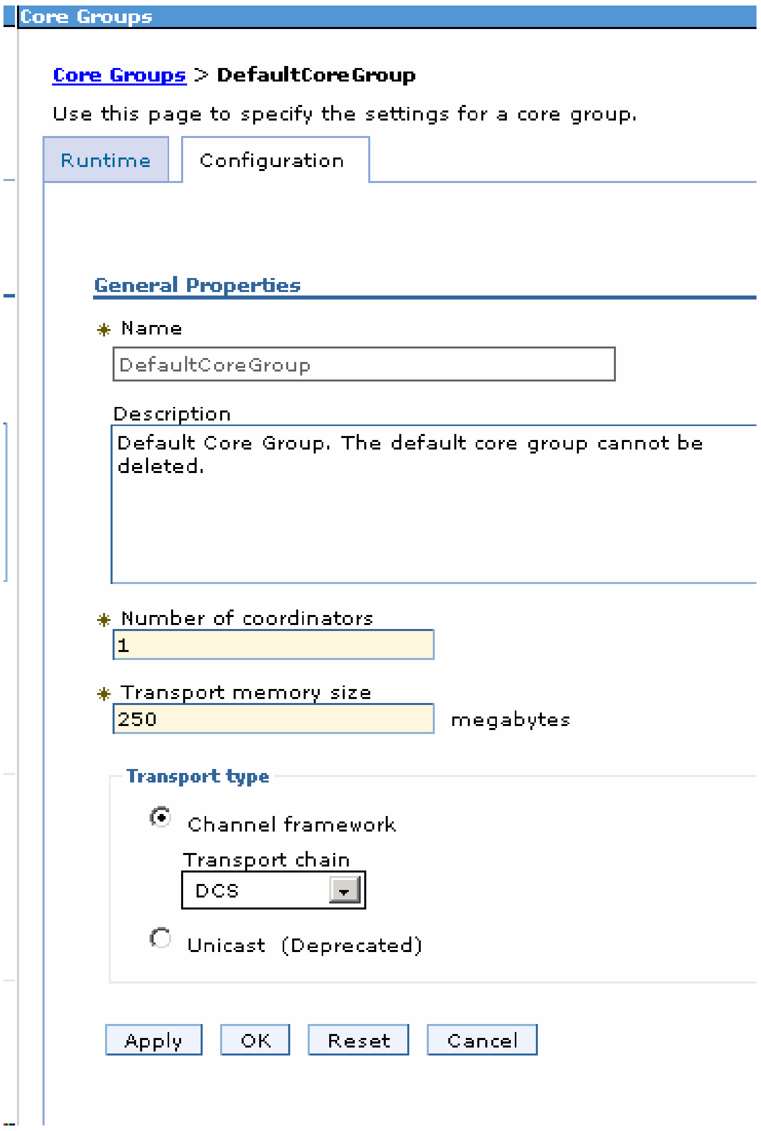
⊞ Preferences

New Delete

Select	Name	Value	Description
You can administer the following resources:			
☐	on.outgoing.message.replication	true	
Total 10			

Figure 11: Custom Property to set the Replication Mode

A high availability parameter that determines the buffer capacity used for exchanging the replicated data is the "Transport memory size" in Figure 12. As shown, the memory buffer size that is specified is 250 Mbytes.


Core Groups

Core Groups > DefaultCoreGroup

Use this page to specify the settings for a core group.

Runtime | Configuration

General Properties

* Name
DefaultCoreGroup

Description
Default Core Group. The default core group cannot be deleted.

* Number of coordinators
1

* Transport memory size
250 megabytes

Transport type
(●) Channel framework
Transport chain
DCS
( ) Unicast (Deprecated)

Figure 12: Setting the high availability buffer size

9 Summary

Enterprises are adding near real-time communications capabilities into existing and new applications by combining the HTTP and the Session Initiation Protocol (SIP). This article has reviewed the performance of a web application that uses both SIP and HTTP protocols. It identified the various tuning parameters and processes for achieving optimal performance of the WebSphere Application Server and Proxy Server.

The example web voice mail application performance demonstrate that the WebSphere Application Server can easily satisfy the performance goals of converged web applications like social networking, unified communications (i.e., integrating instant messaging, email, phone communication, etc.), and web based self-serve. Using the example application, a single HS21 blade server's achieved 931 converged operations per second which is well beyond what most enterprises would require. If this high level of capacity is not needed then additional SIP or HTTP applications could be deployed on the same hardware for greater efficiency.

Although a single hardware server was used for the non-HA measurements, the performance results can be extrapolated to additional servers because the WebSphere Application Server scales out horizontally to large deployments.

WebSphere Application Server high performance extends into high availability configurations. The built in high availability of WebSphere Application Server provides excellent capacity, achieving 349 converged operations per second per HS21 blade server. The High Availability measurement results can also be horizontally scaled out to larger deployments. However, some care must be taken to make sure that backup servers do not run on the same hardware as their primary server.

Other IBM WebSphere products that were not included in this example can enhance these capabilities. The Communications Enabled Applications feature pack enables developers to build communications enabled applications using Web 2.0 widgets and services, without requiring in-depth technical knowledge about SIP. This would include adding capabilities to web applications like: click to call, co-browsing, and call notifications to web applications.

WebSphere eXtreme Scale can improve high availability by providing a distributed, transactional, highly available cache which can be used for geographic redundancy. It can also be used to build high throughput transaction processing applications that may require web and/or voice multi-modal interactions.

If external factors require tightening up the latency values or having smarter management of retransmissions then WebSphere Real-Time for Linux can help to make the latency more deterministic and/or WebSphere Virtual Enterprise can use its autonomic admission controls to guard against extreme load or Denial of Service attacks.

10 Appendix: Tuning Information

The following table summarizes the tuning information that was covered in the previous sections.

Category	Command or Value	Comment

Network interface card tuning	`ethtool -s <eth int> autoneg off`
	`/sbin/ifconfig <eth int> txqueuelen 2000`
	`ethtool -s <eth int> speed 1000`
	`ethtool -s <eth int> duplex full`
	`ethtool -A <eth int> rx on tx on`
	`ethtool -C <eth int> adaptive-rx off adaptive-tx off rx-usecs 20 rx-frames 5 tx-usecs 60 tx-frames 11`
	`ethtool -G <eth int> rx 511 rx-jumbo 255 tx 511`
		The <eth int> is the name of the interface, such as ‘eth0’.

Linux operating system tuning	`chkconfig nscd on`	Start the name server caching demon.

	`echo 16777216 > /proc/sys/net/core/rmem_max`
	`echo 2097152 > /proc/sys/net/core/rmem_default`	

Category	Command or Value	Comment

	`echo 16777216 > /proc/sys/net/core/wmem_max`
	`echo 2097152 > /proc/sys/net/core/wmem_default`
	`echo 10000000 > /proc/sys/net/core/optmem_max`
	`echo 4096 87380 16777216 > /proc/sys/net/ipv4/tcp_rmem`
	`echo 4096 65536 16777216 > /proc/sys/net/ipv4/tcp_wmem`
	`echo 8388608 8388608 8388608 > /proc/sys/net/ipv4/tcp_mem`
	`echo 400 > /proc/sys/net/unix/max_dgram_qlen`
	`echo 400 > /proc/sys/net/core/message_burst`
	`echo 2800 > /proc/sys/net/core/mod_cong`
	`echo 1000 > /proc/sys/net/core/lo_cong`
	`echo 200 > /proc/sys/net/core/no_cong`
	`echo 2900 > /proc/sys/net/core/no_cong_thresh`
	`echo 3000 > /proc/sys/net/core/netdev_max_backlog`	

	`ulimit –n 16000`	Increase the number of file handles to 16,000.

	Add the following lines to the file /etc/modprobe.conf :
	`alias net-pf-10 off`
	`alias ipv6 off`
		Disable IPv6 which can slow down communications.

Application server JVM option tuning	Menu path is: “Application severs” → <application server name> → “Java and Process Management” → Process definition → Java Virtual Machine. `-Xtgc:parallel -Xdisableexcessivegc -Xmn150m -Xgcpolicy:gencon -Xgc:scvNoAdaptiveTenure,scvTenureAge=1,stdGlobalCompactToSatisfyAllocate –Xloaminimum0.01 –Xcompactexplicitgc -XX:MaxDirectMemorySize=128000000`	
Application server JVM heap size tuning	Menu path is: “Application severs” → <application server name> → “Java and Process Management” → Process definition → Java Virtual Machine. Initial heap size `1536` MB Maximum heap size `1536` MB	
Proxy server JVM options tuning	Menu path is: “Application severs” → <application server name> → “Java and Process Management” → Process definition → Java Virtual Machine. `-Xmo160m -Xgcpolicy:gencon -Xgc:noAdaptiveTenure,tenureAge=8,stdGlobalCompactToSatisfyAllocate -Xtgc:parallel -XX:MaxDirectMemorySize=128000000`	
Proxy server JVM heap size tuning	Menu path is: “Application severs” → <application server name> → “Java and Process Management” → Process definition → Java Virtual Machine. Initial heap size `500` MB	

Category	Command or Value	Comment
	Maximum heap size `650` MB	
Application server high availability tuning	Menu path is: "Application Servers"→<application server name> → "SIP container" → "Custom properties". `end.of.service.replication=true`	Custom properties panel for the application server's SIP container
	Menu path is: "Core Groups" → "DefaultCoreGroup". Transport memory size = 250 megabytes	This setting controls the peak amount of dynamic memory that can be used for caching the data replication messages. The default value for this property is 50 megabytes.

Application server communication tuning	Menu path is: "Application severs" → <application server name> → SIPCInboundDefaultUDP → UDP inbound channel (UDP 1) → Custom property.
	`receiveBufferSizeSocket=3000000`
	`sendBufferSizeSocket=3000000`
		Buffer size for the UDP socket.

Table 3: Summary of SIP and HTTP Tuning Parameters for Linux

11 Resources

[CMG-Sip] C. Hrischuk, G. DeVal. "A Tutorial on SIP Application Server Performance and Benchmarking", Computer Measurement Group 2006. www.cmg.org/conference/cmg2006/awards/6084.pdf

[JeeTuning] Christopher Blythe and David Hare. "Case study: Tuning WebSphere Application Server V7 for performance." http://www.ibm.com/developerworks/websphere/techjournal/0909_blythe/0909_blythe.html

[GcGencon] http://www.ibm.com/developerworks/java/library/j-ibmjava2/

[GcPerf] http://www.ibm.com/developerworks/library/i-gctroub/

[Info1] "Tuning SIP servlets for Linux" http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tsip_tunelinux.html

[JvmDiagnostic] http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html

[SIPp] See http://sipp.sourceforge.net/doc/

[64BitJVM] “IBM WebSphere Application Server V7 64-bit Performance” https://www14.software.ibm.com/webapp/iwm/web/signup.do?lang=en_US&source=sw-app&S_PKG=WAS_V7_64-bit&S_TACT=109DA54W

[CEA] http://publib.boulder.ibm.com/infocenter/wasinfo/fep/index.jsp?topic=/com.ibm.websphere.ceafep.multiplatform.doc/info/ae/ae/ccea_jsr289_overview.html