The Association of System Performance Professionals

The **Computer Measurement Group**, commonly called **CMG**, is a not for profit, worldwide organization of data processing professionals committed to the measurement and management of computer systems. CMG members are primarily concerned with performance evaluation of existing systems to maximize performance (eg. response time, throughput, etc.) and with capacity management where planned enhancements to existing systems or the design of new systems are evaluated to find the necessary resources required to provide adequate performance at a reasonable cost.

This paper was originally published in the Proceedings of the Computer Measurement Group's 2006 International Conference.

For more information on CMG please visit http://www.cmg.org

A TUTORIAL ON SIP APPLICATION SERVER PERFORMANCE AND BENCHMARKING

Curtis Hrischuk, Ph.D.
WAS SIP and IMS Lead Performance Engineer, IBM Software Group

Gary DeVal
WAS SIP Performance Engineer, IBM Software Group

31st Annual International Conference of The Computer Measurement Group, Inc.
December 3 - 8, 2006
Reno, Nevada USA

The Session Initiation Protocol (SIP) is an internet protocol for establishing sessions between two or more parties. It is becoming ubiquitous in uses such as Voice over IP, instant messaging, Internet TV, and others. The Java community has even provided a standardized API so that SIP applications can now be built within J2EE application servers. These new capabilities also bring with them new performance engineering methods, tools, and benchmarking needs. This paper describes the experiences and processes for the performance engineering of SIP applications in a J2EE environment.

1. Introduction

The Session Initiation Protocol (SIP) [1] is used for negotiating sessions between two or more parties that want to interact or communicate. SIP is used for Voice over IP (VoIP) or instant messaging applications to connect parties that want to exchange data, audio, or video. SIP differs from other approaches to session negotiation because it is decentralized, moving the control handshaking to the end points rather than the control being centralized. This makes it extensible, scalable, and useful for mobile applications. SIP is becoming ubiquitous. Recently, application servers have been developed which interact using the HTTP and SIP protocols for the same application; referred to as *converged applications*. It is expected that the next generation of web based applications will be converged. SIP has performance characteristics that differ from HTTP in several respects. So, it is important for the performance engineer to understand those differences and how to negotiate them.

This paper is intended to be a tutorial about SIP and its performance characteristics. The paper is structured as follows. SIP is relatively new so an overview is presented first. This is followed by a discussion of tuning Linux to support SIP traffic. Then a SIP application performance engineering methodology is presented.

2. SIP Overview

Comparing SIP to the popular HTTP protocol is the logical place to start. SIP is similar to HTTP in the following ways: SIP requests and responses are text, looking very similar to HTTP/1.1. Like HTTP, the requests and responses can contain *message bodie*s: typically session descriptions. An example SIP request is shown in Figure 1.

Just like HTTP, SIP uses a URI to uniquely identify a user, device or service. In Figure 1, the URI is “sip:alice@domain2.com”. Also, a servlet lifecycle specification has been developed for SIP, much like the HTTP servlet definition. This enables SIP applications to be developed within a common framework for deployment in hosting environments called SIP Servlet Containers, analogous to HTTP Servlet Containers.

SIP differs from HTTP in several respects:

- It has QoS latency requirements on protocol responses;
- It has a peer to peer architecture, rather than the HTTP client-server approach;
- A SIP servlet may also act as a SIP client;
- Transactions and sub-transactions may be stateless or stateful;
- Interactions are not only synchronous (like HTTP) but may also be asynchronous;
- More than one response may be generated in response to a single request; and
- A single SIP request may be served by multiple SIP applications in the same application server.

These characteristics of SIP provide a much more flexible and versatile application framework than HTTP, and can therefore be used to overcome some of HTTP’s limitations.

```
INVITE sip:alice@domain2.com SIP/2.0
Via: SIP/2.0/UDP 9.150.21.198:5060;rport;branch=z9hG4bK023C1
From: sip:bob@domain1.com;tag=2968942665
To: sip:alice@domain2.com
Contact: sip:softphone@9.150.21.198:5060
Call-ID: 1403092@192.168.0.103
CSeq: 1001 INVITE
Max-Forwards: 70
Content-Type: application/sdp
Content-Length: 240

v=0
o=17476002050 64593273 64593313 IN IP4 9.150.21.198
c=IN IP4 9.150.21.198
t=0 0
m=audio 8000 RTP/AVP 0 8 101
a=rtpmap:0 pcmu/8000
```

Figure 1: SIP Invite Request

1.1 SIP Example Message Exchange

An example exchange between two SIP end-points (User Agents -UA) is shown in Figure 3. In this example, bob@domain1.com is trying to place a VoIP phone call with alice@domain2.com. Bob's UA begins the call by sending the INVITE SIP message which traverses the network using intermediate SIP Proxy servers to eventually reach Alice's UA. Rather than wait for the target end point to send back a response, the Proxy servers send the intermediate SIP response TRYING to indicate that effort to reach the target is occurring. The target end point (Alice's UA) responds that the phone is RINGING and, finally, a 200OK response is sent when Alice picks up the phone. Bob's UA then sends the acknowledgement (ACK) to tell Alice's UA that the call can proceed. It is important to emphasize that the actual media data traffic (e.g., the voice conversation) does not use the SIP protocol but other protocols for that purpose, such as the Real Time Protocol [2]. Finally, Alice hangs up the phone so the BYE message is sent to Bob, followed by Bob's OK message to acknowledge receipt of the BYE.

The SIP exchange of Figure 3 illustrates the use of several of the SIP message types, called methods. There are several other SIP methods that can be used for various purposes and they are listed in Table 1 for completeness.

1.2 SIP Timing Requirements

Like many other protocols, SIP has QoS timing requirements for when a response is expected. As shown in Figure 2, the sender of the INVITE message expects a TRYING response within 500 milliseconds or it is assumed that the INVITE message was lost in transmit and the INVITE is retransmitted. There are several additional timers that govern the SIP protocol [1] and they are shown in Table 2. A key performance goal is to not exceed the timer values because this can have significant consequences where tens of thousands of requests are retransmitted.

Method	Description
INVITE	initiate call
ACK	confirm final response
BYE	terminate (and transfer) call
CANCEL	cancel searches and "ringing"
OPTIONS	query recipient's supported features and availability
REGISTER	register with location service
INFO	mid-call information (ISUP)
COMET	precondition met
PRACK	provisional acknowledgement
SUBSCRIBE	subscribe to event notification
UNSUBSCRIBE	cancel event notification
NOTIFY	notify subscribers
REFER	ask recipient to issue SIP request (call transfer)
MESSAGE	instant message body transport

Table 1: SIP Protocol Methods

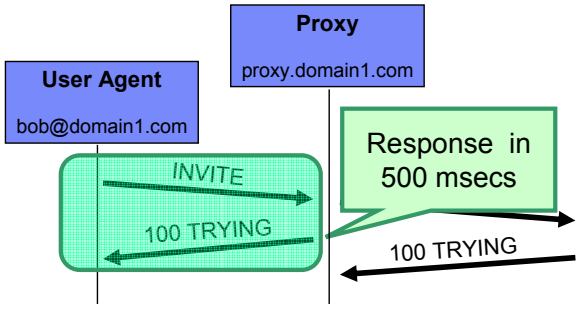

Figure 2: SIP Timing Constraints

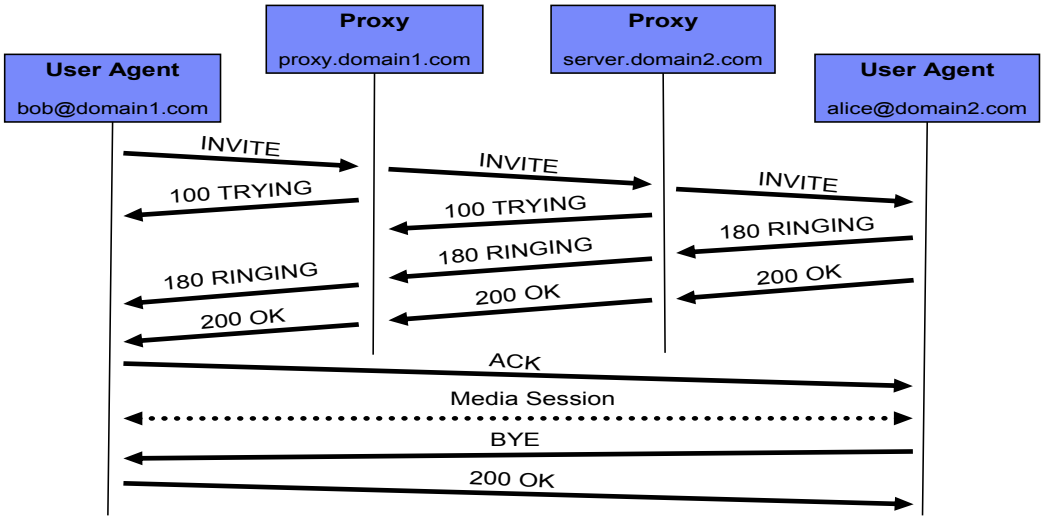

Figure 3: Example SIP Exchange

Timer	Default value	Meaning
T1	500 ms	Round-trip time (RTT) estimate
T2	4 sec.	Maximum retransmission interval for non-INVITE requests and INVITE responses
T4	5 sec.	Maximum duration that a message can remain in the network
Timer A	initially T1	INVITE request retransmission interval, for UDP only
Timer B	64*T1	INVITE transaction timeout timer
Timer C	> 3 min.	Proxy INVITE transaction timeout
Timer D	32 sec. for UDP 0 sec. for TCP and SCTP	Wait time for response retransmissions
Timer E	initially T1	Non-INVITE request retransmission interval, UDP only
Timer F	64*T1	Non-INVITE transaction timeout timer
Timer G	initially T1	INVITE response retransmission interval
Timer H	64*T1	Wait time for ACK receipt
Timer I	T4 for UDP 0 sec. for TCP and SCTP	Wait time for ACK retransmissions
Timer J	64*T1 for UDP 0 sec. for TCP and SCTP	Wait time for retransmissions of non-INVITE requests
Timer K	T4 for UDP 0 sec. for TCP and SCTP	Wait time for response retransmissions

Table 2: SIP Protocol Timers

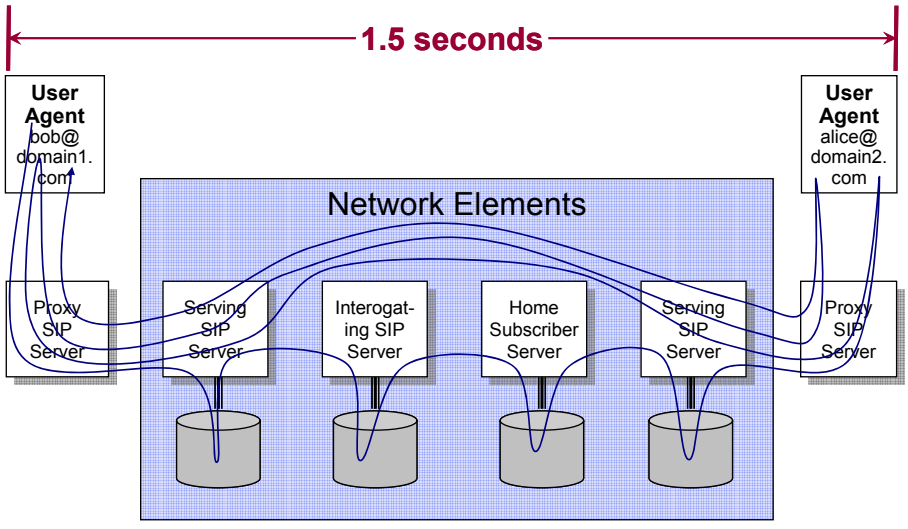

Figure 4: Example VoIP Network Traversal (Signalling Plane)

It is also useful to view how a VoIP call is set-up, traversing the network.

is such a network traversal, where several SIP (proxy) servers forward the SIP messages. As shown in the diagram, several interactions with intermediate servers may be required to establish a VoIP call.

POTS (traditional voice network) timing requirements allow 1.5 seconds for the call to be established (from the initial INVITE to destination RINGING). This is a tight requirement for several reasons: there are tens of thousands of users competing for these resources; network latency slows the communication; queuing occurs at the servers; database accesses are needed to look-up information.

Clearly, SIP performance requirements are challenging.

1.3 Using SIP for Converged Applications

To make SIP easy to use, a Java application programming interface (API) similar to HTTP servlets has been devised, called JSR 116 [3]. This API provides an easy-to-use SIP programming model. Converged applications that can interact via the HTTP and SIP protocols can be enabled with an architecture similar to that shown in Figure 5 [4].

The beauty of this design is that an application can be multi-modal with the same application context shared by the HTTP portion of the application as well as the SIP portion of the application. This means that only one converged application need be written for the two protocol types.

To understand the power of the converged application, a simple application description may be useful. Imagine you are trying to repair some appliance so you bring up the manufacturer's web site. You enter the model number, year, and other relevant information into a web form and submit the form. Based on your input, a database is queried to identify the technician most apt to assist you in resolving the issue that is currently free to take a phone call. When the search results are returned, you are offered the option of calling that technician to discuss the situation for the small fee of 50 cents. You choose the option and are directly connected to the technician, conversing with them using your PC's microphone and speaker!

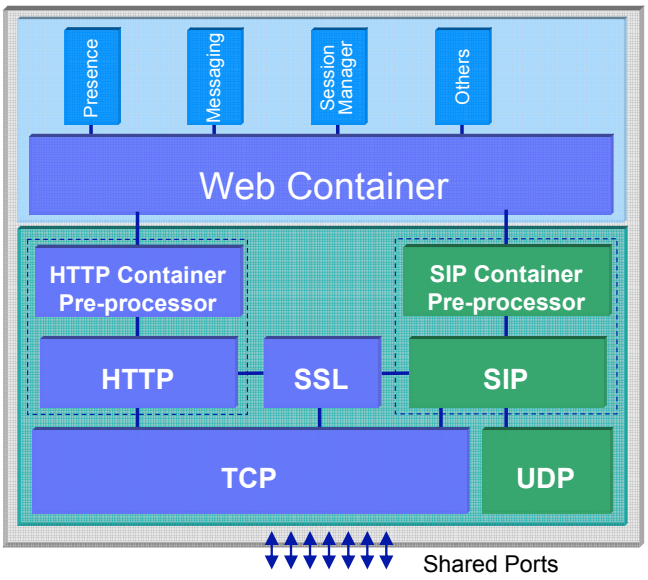

Figure 5: Converged Application Architecture

3. SIP Stack Tuning

As suggested in Table 2, SIP can be used with several transport protocols but the most commonly used protocol is UDP. The benefit of UDP is that there is low state overhead (i.e., no need to set up a TCP connection) and efficient processing of the (usually) small SIP packets. TCP is normally used where encryption (e.g., SSL) is required or the SIP message size exceeds the MTU of UDP. Since the majority of SIP traffic is over UDP, tuning the server for UDP communication will result in fewer lost packets and reduce the overall latency.

There are three levels at which the SIP communication stack tuning occurs. The most obvious is at the kernel level by increasing the number of communication buffers. However, the NIC driver tuning is also important so that bursts of SIP traffic do not result in delays and retransmission of messages. The NIC tuning increases the buffer capacity and decreases the interrupt interval to move the packet from the NIC memory to the kernel and then into the user space. Lastly, general network tuning needs to be applied. The examples given in this section are for the Linux 2.6 kernel with lessons learned from the Linux 2.4 kernel as well.

1.4 General Network Tuning

The first tuning step is to remove unused network interfaces. Typically, a server will have two or four network interfaces. Through trial and error, it was found that there was a large performance hit on the active network interface if the unused network interface remained in the active profile. This was

unexpected since disabling an interface should be sufficient but the measurements showed that removing an unused interface was required. This problem was diagnosed using a simple FTP throughput test to measure the network bandwidth between servers. Very low throughput was observed until the unused network interfaces were removed.

The second general network tuning optimization was to disable the auto-negotiate Ethernet setting on the servers and the Ethernet switch that the servers were connected to. This problem was again diagnosed using a simple server-to-server FTP test with a large file. A paltry throughput of 5 Mbps was observed with speeds as low as 13 Kbps. This was resolved by forcing the Ethernet values to: 1000 Mbs, full duplex, on both. When either side was set to auto-negotiate speed there were performance problems.

These two changes increased the SIP message throughput by 748 SIP messages per second which is about 25%.

1.5 NIC Driver Tuning

The NIC driver tuning is very low level. The underlying goal is to change the parameters so that a received packet is removed from the NIC's memory as soon as possible. This reduces message processing latency at the expense of low amortization of the CPU processing overhead (e.g., cache flushing). The parameters for this are:

- adaptive-rx, adaptive-tx: adaptively adjust interrupt intervals based on packet rates. This should be disabled.
- rx-µsecs: how many µsecs to delay an RX interrupt after a packet arrives. A low value is preferred.
- rx-frames: how many packets to delay an RX interrupt after a packet arrives. A low value is preferred.
- tx-µsecs: How many µsecs to delay a TX interrupt after a packet is sent. A low value is preferred.
- tx-frames: How many packets to delay a TX interrupt after a packet is sent. A low value is preferred.

An example Linux command to set up the NIC interface parameters is 'ethtool -C eth0 adaptive-rx off adaptive-tx off rx-usecs 20 rx-frames 5 tx-usecs 60 tx-frames 11'.

Another tuning step is to set up the NIC receive and transmit ring parameters, which should be increased to the largest values possible. The Linux command to do this is "ethtool -G eth0 rx 511 rx-jumbo 255 tx 511'.

1.6 Kernel Tuning

The Linux tuning parameters for the SIP stack are shown in Table 4. There is much literature that is available so no further explanation is provided here. Also use the /sbin/ifconfig command to change the maximum size of the interface's transmit queue (e.g., '/sbin/ifconfig eth0 txqueuelen 2000').

1.7 Results of SIP Stack Tuning

The MGEN tool (http://mgen.pf.itd.nrl.navy.mil/) was used to select a NIC driver and to explore the UDP performance of the tuning modifications. The Multi-Generator (MGEN) is open source software by the Naval Research Laboratory (NRL) PROTocol Engineering Advanced Networking (PROTEAN) Research Group. MGEN provides the ability to perform IP network performance tests and measurements using UDP/IP traffic.

MGEN was used to select from the three NIC drivers that were available at the time. The three drivers were:

- The original tg3 driver in RHEL4, version 3.27-rh
- A new tg3 driver from Broadcom, version 3.43b
- A NIC driver from Broadcom, version 8.3.14

The performance of each of these drivers was examined using bidirectional UDP traffic between two test systems that were connected using a Gigabit Ethernet switch. The selected packet size is 338 bytes which was found to be the average SIP packet size. The results of the comparison are shown below in Table 3, with the new tg3 driver providing the best performance with the lowest packet loss rate at a high message rate. For this reason, the new tg3 driver was selected.

Packet size: 338 bytes		**Percent Packet Loss**		
Message Rate (1,000 pkts / sec)	**Bit rate** (Mbps)	**Original tg3 driver**	**New tg3 driver**	**Bcm5700 driver**
10	27	-	-	-
20	54	-	-	-
30	81	0.1	0.0	0.0
40	108	-	0.0	-
50	135	0.0	0.0	0.1
60	162	13.3	1.0	0.8
70	189	15.5	9.0	12.1
80	216	18.8	11.9	17.0

Table 3: NIC Driver Packet Loss Comparison

The next set of measurements was to continue the NIC and kernel tuning, measuring the benefit that it produced. As shown in Table 5, there a is significant improvement in the processing of UDP traffic from the tuning, at all message rates.

Kernel Parameter	Description
rmem_max	maximum receive window
rmem_default	default receive window
wmem_max	maximum send window
wmem_default	default receive window
optmem_max	socket option memory allocation limit
tcp_rmem	Receive buffer allocations
tcp_wmem	Send buffer memory allocations
tcp_mem	TCP memory usage limits in pages
max_dgram_qlen	maximum data queue length per socket in packets/datagrams
message_burst	Burst rate limit for log messages
hot_list_length	max length of per cpu packet buffer allocation list
mod_cong	Receive queue lengths below this are only moderately congested
lo_cong	Receive queue lengths below this have low level congestion
no_cong	Receive queue lengths below this are not congested
no_cong_thresh	Low water mark for re-starting congested devices
netdev_max_backlog	Maximum receive packet backlog before congestion control is enforced

Table 4: Linux 2.6 SIP Stack Tuning

Message Rate (1,000 pkts / sec)	Bit rate (Mbps)	New tg3 Driver Percent Packet Loss		Delta
		Without tuning	With kernel and NIC tuning	
10	27	0.16%	0.00%	-0.16%
20	54	0.29%	0.00%	-0.29%
30	81	0.43%	0.02%	-0.41%
40	108	2.18%	0.01%	-2.18%
50	135	7.23%	0.05%	-7.18%
60	162	18.06%	1.02%	-17.04%
70	189	25.81%	8.96%	-16.85%
80	216	31.50%	11.92%	-19.59%

Table 5: NIC and Kernel Tuning Benefits

2 A SIP Performance Engineering Process

There are three areas that SIP application performance needs to be concerned with:

- Latency through the application: Unlike HTTP, SIP has sub-second timing requirements that affect the health of the system. There are many sources of latency in a SIP application, including SIP stack processing time, SIP stack queuing, virtual machine garbage collection latency, thread scheduling, etc.
- SIP message throughput (server): The rate at which SIP messages need to be processed must be faster than the input rate of offered traffic. This is governed by the latency factors above, as well as the CPU and network capacity.
- Session capacity per node: SIP is a stateful application so, for each SIP user interaction, there is session state that is persisted for the duration of the call. SIP calls may have a duration of six or more minutes and be offered at a rate of hundreds of calls per second, translating to tens of thousands of active sessions at any given time. The result is that a significant amount of memory is consumed in maintaining the session state. Since most operating systems have a limit on the amount of data memory that a process may access, each SIP application process will be able to support some maximum number of SIP sessions.

These three areas have several input and output factors that go into devising the architecture of an application and the equipment needed to meet the capacity requirements. A performance engineering process for SIP needs to keep each of these areas in focus. When conducting performance tests, these factors, as well as others, need to be monitored. This is all examined in the following sub-sections.

2.1 SIP Performance Input Factors

There are many input factors into the performance of a SIP application. The most important factors are:

- Application latency budget: As shown in Figure 2, there are SIP timing constraints that the SIP stack and application needs to honor. For engineering purposes, a latency budget is allocated which is the mean delay with a standard deviation as well

as a 99.99 percentile delay through the application. Usually, the main factor in this latency would be garbage collection time in a virtual machine based application (e.g., Java).
- System garbage collection latency budget: As shown in Figure 4, a SIP application is one element in the entire system. To engineer the system, a SIP latency budget is allocated to each element in the system. This latency budget is typically more flexible than the application latency budget because there is some elasticity in the system.
- System configuration: Depending on the workload, a single server may be adequate or a cluster of servers may be needed. Another significant option is whether a high availability configuration is needed to avoid a single point of failure. High availability introduces more activity in a VM-based application because, in addition to the regular application objects, state objects are replicated to copy state information from a working VM to a backup VM.
- Type of application: There are several canonical architectures for SIP applications (e.g., proxy, Back-to-back user agent) and each has its own performance profile.
- Maximum SIP message rate: Clearly, a major workload factor is the rate at which SIP messages are presented to the application and the application's processing rate must exceed this value.
- Maximum session creation rate: The rate of memory consumption is governed by the session creation rate. Session creation rate also affects CPU utilization due to the memory management activity required for new session objects.
- Duration the session exists: Each user session can expire due to a default session time-out value in the sip.xml file of the application or if the end-points terminate the session. Session duration, session creation rate, and per-session memory consumption drive working memory requirements for all system components which maintain session state.
- Maximum application session memory consumption: This is the amount of memory consumed when a user session is created and during the lifetime of the user session. This is usually specified as the KBytes per application session.
- Average CPU utilization: For operational purposes, it is useful to specify a budget for the CPU consumption. This is typically less than 100% to allow for management and operating system activity.
- Transport type: As mentioned, UDP is the primary SIP transport. TCP is an option if SIP messages are too large or data encryption is needed for security. UDP and TCP have differing performance profiles.
- Authorization: In some cases, a SIP application may have to perform authentication and authorization, which adds performance overhead to the system.

These factors help to characterize the overall behavior of the application. Various environmental factors (discussed next) must also be considered to engineer SIP application performance.

2.2 SIP Performance Environment and Tuning Factors

Deployment and tuning of a SIP application defines an execution environment. Key execution environment performance factors are:

- Number of servers: If the maximum SIP message rate or maximum session creation rate exceeds that of a single server, then a cluster of servers is usually configured;
- Number of CPUs per server: The more processors, the faster the processing of the SIP stack and other maintenance activities (e.g., VM garbage collection);
- Number of SIP application processes per server: A single SIP application process may not have enough memory to support the rate at which sessions are created and the duration for which those sessions exist; and
- Virtual machine tuning values: If a virtual machine environment is used for the SIP stack or SIP application, there are many tuning factors.

These are the key environmental factors.

In particular, a virtual machine environment [5, 6], provides many tuning opportunities that affect latency, message throughput, and memory consumption. Most of this tuning revolves around reducing garbage collection latency which affects all of the key performance areas for SIP applications. Those options that are important for SIP are:

- Heap size: The heap size is the working memory for creating SIP and application objects. It limits the number of concurrent SIP sessions. It is also important because it affects the garbage collection latency; in general, the larger the heap the longer the garbage collection latency. A starting heap size of 800 MBytes is reasonable.
- Garbage collection algorithm: There are many garbage collection options available for each VM environment and care must be taken to select the one that has the lowest latency and/or most deterministic latency during the application execution. Recent generational garbage collection algorithms have very good performance.

- Number of concurrent, foreground GC threads: When the amount of free memory on the heap reaches a low level, a garbage collection occurs which takes control of all of the available CPUs in order to perform a garbage collection in the shortest possible time.
- Number of concurrent, background garbage collection threads: During the execution of the application, there are idle CPU periods where background garbage collection threads can perform preparatory work, e.g. marking dead (unused) objects, in order to reduce the required execution time when a garbage collection has full use of the CPUs.
- Object allocation tax rate: Another technique to reduce the latency of a "stop the world" garbage collection is to perform a little garbage collection activity each time more memory is used from the heap. This amortization of the garbage collection is not done by threads specific to garbage collection but on the threads that execute on behalf of the application. This can be thought of as an object allocation tax.

Each VM environment has additional tuning options and it is suggested to consult their literature.

2.3 SIP Performance Engineering Process

Managing all of these various factors that affect performance requires a methodical approach. The SIP performance engineering process that has evolved for our purposes is shown in

Figure 6. It begins with a performance test specification that identifies the key factors for the SIP application (see previous sub-sections) since not all factors have equal weight. A practical requirement for efficiency and repeatability is to have an automated test execution framework that will perform a load test on the system and collect the results. Then a test script can be generated to execute multiple measurement tests on the SIP application, systematically exploring the various input, environment, and tuning factors. Some form of post-execution automated analysis is necessary because of the large volume of data collected. A comparison analysis of several different measurement tests within the same test specification, across different SIP applications, or different system configurations is very useful – automation is again recommended here. Finally, a proposal to enhance the performance, or tune the system, is the result.

The automated test execution that was devised for our purposes, affectionately referred to as *Convergence Automation with BASH (CAB),* was based on BASH shell scripts that followed important programming conventions.

The CAB design had several key goals: it would encapsulate and encode best practices, such as tuning operations; capture all relevant information for performance diagnostics; use secure communication (e.g., SSL) between the system components; have minimal dependencies with no special languages or tooling (i.e., the console is your IDE); use defensive programming (i.e., use *assert* to know if something went wrong); build debugging support into all steps (i.e., log everything); and provide an incrementally extensible platform.

The structure of the Convergence Automation with Bash (CAB) framework is shown in

Figure 7. In this configuration, the *Controller* controls the execution of the *Load Driver* and the *System Under Test*. The *Test Configuration File* has the information needed to execute the test on the system. The *System Configuration Files* provide parameter values for CAB so that it is easy to retarget and to maintain CAB. The *Personal Configuration Files* list those parameter values that cut across different configurations or test runs. For flexibility, a test parameter could exist in multiple files or have a different value specified on the command line. The first occurrence in the search for the parameter would be used. The different configuration files were searched in the following order to minimize changes to configuration files: (1) command line parameters; (2) Personal Configuration File; (3) Test Configuration file; (4) the System Configuration files that are installed with CAB; and (5) default configuration values specified in the scripts.

The general behavior of a SIP measurement test is as follows:

1. Describe the system (i.e., Load Driver(s) and System Under Test (SUT)) and test parameters in a test configuration file. Additional system configuration files are provided with CAB for default values or customization of CAB.
2. Start the control script, using command line options that identify the type of test to execute, the configuration of the system, as well as other parameters that are relevant to each execution (e.g., SIP calls per second). The control script then completes the remaining steps.
3. Stop all the SUT servers in the SUT, clear out their log files, and reboot the servers.
4. Start the SUT servers.
5. The Load Driver will generate a small load against the SUT to warm it up. This warm up test allows for classes to be loaded, the VM Just In Time (JIT) compiler to compile

frequently used classes, as well as other start up operations.[1]

6. Measurement tools begin running on the Load Driver and the SUT servers to capture resource usage.
7. The Load Driver generates the specified load against the SUT for a specified period of time (referred to as the *test1* period).
8. When the test period ends the measurement tools and the Load Driver are stopped.
9. The measurement data, log files, and other relevant information is gathered and stored in a zip file. The zip file can be uploaded to the options *File Server* for publishing or archival purposes.

Many variations to this general flow are possible using command line options for flexibility.

CAB has been designed to be able to modify all identified environment factors, input factors, and tuning factors. In a period of three months, 1,103 stored experiments were performed that generated 23.4 GBytes of *compressed* data.

Diagnosing SIP performance issues is difficult because it has a time sensitive component that can be sub-second in nature. Over time, the collected data evolved to include: load driver logs, SIP application server logs, SIP application server configuration files, servlet configuration files, any error files (e.g., core, heapdump, failure event reports, javacore), tuning scripts, module information, hardware information, operating system information, and resource measurement data (e.g., NMON [7]). In essence, any information that would be necessary to completely define the test scenario was collected, aside from the application binary libraries. This facilitates precise analysis of observed SUT behaviors, as well as accurate test reproduction.

The default behavior during test execution was to also capture SIP packet traffic using Ethereal [8] just before and just after the steady state measurement period. This captured traffic was analyzed to understand if SIP messages were being retransmitted or if SIP protocol errors were occurring. Optionally, the captured SIP packets could also be stored in the zip file for detailed analysis. Note that SIP packets were not captured during the steady state measurement period because packet capture is disk and CPU intensive, and thus would perturb the system under test and invalidate resource measurements.

The default behavior also recorded ping activity between several servers to monitor the network connectivity. Like all networks, occasional disruptions of network connectivity arise that cause interruptions in SIP communication and application processing. By recording the ping activity it was possible to quickly identify if an issue was caused by the network or the application.

2.4 Methodology for Analyzing SIP Performance

Once the data is collected, the analysis can begin.

The first analysis step is to pre-screen the input data to determine if the measurements are valid. The SIP statistics are generated from the Ethereal's analysis of the captured SIP traffic and indicate if there were any protocol errors or retransmissions. Examine the SUT log files or other failure indicators for processing exceptions. Exceptions should be examined to determine whether they related to the test directly or may have been peripheral or unrelated to the functionality being tested. The data in the log files may be informational, i.e. they do not usually indicate severe problems (those show up in the next step) and provide useful clues if functional problems occurred during the test run. It is possible that extreme situations occurred resulting in thread dumps, heap dumps, and Java core files which would indicate an unsuccessful test and problems. Exceptions directly related to the functionality being tested may indicate functional or performance problems which can invalidate performance analysis of the test results.

The next step is to process the input data for a single run, which takes two forms. The first analysis form is as time series graphs to be able to review the behavior of key parameters at a glance. This is important because generating statistics from the measurements can be misleading or fail to identify other issues. As an example,

Figure 8 is the time series graph for the CPU utilization of the SUT while under a fairly constant load. What is unusual is that the CPU utilization follows a square wave for no apparent reason – a fact that would not be evident from a calculation of the mean CPU utilization or standard deviation value. After much investigation it was discovered that the Linux Name Service Caching Daemon (nscd) was not started and this was the source of the problem. Since nscd was not running, every SIP message request initiated a DNS request to lookup an IP address from a given hostname.

[1] A test variation is to start the SUT servers while the warm up load is occurring. This was done so that the garbage collection activity of all of the servers would not be synchronized. For example, if the load was applied to all of the servers at the same time, then the load balancer would ensure each server had the same call rate applied so their rate of object creation and death is the same, resulting in each server performing a garbage collection at the same time.

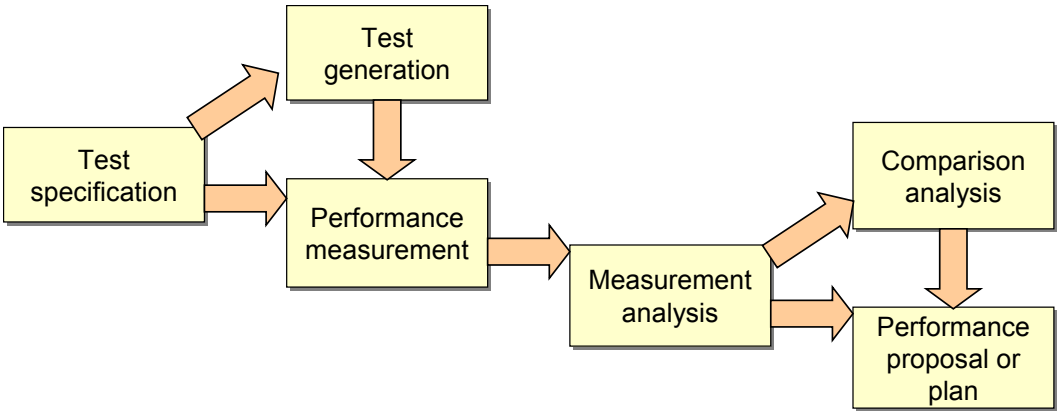

Figure 6: SIP Performance Engineering Process

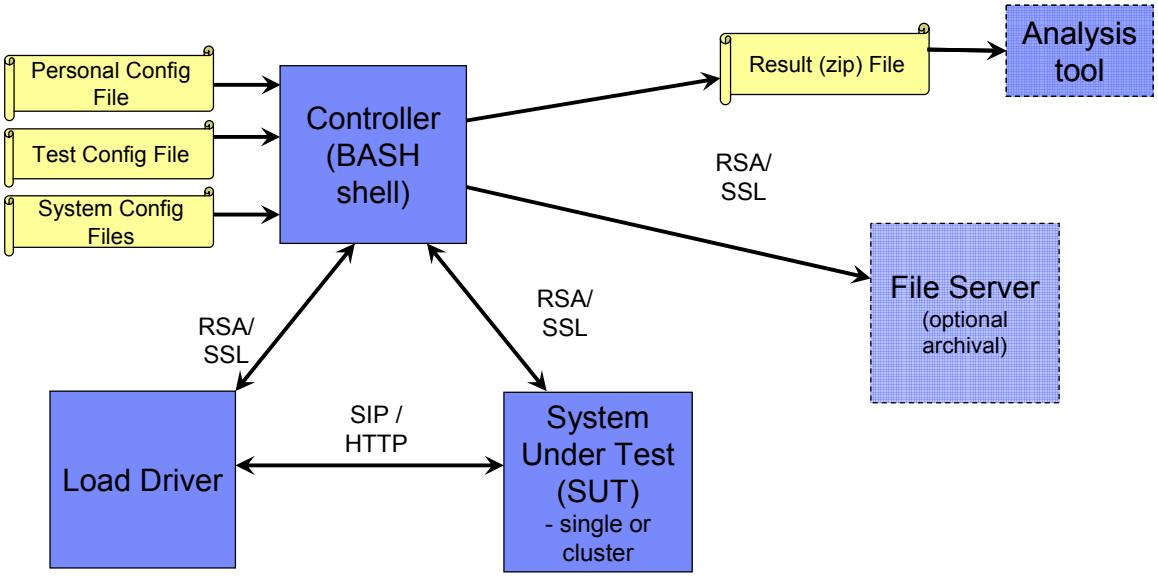

Figure 7: Measurement Automation Framework

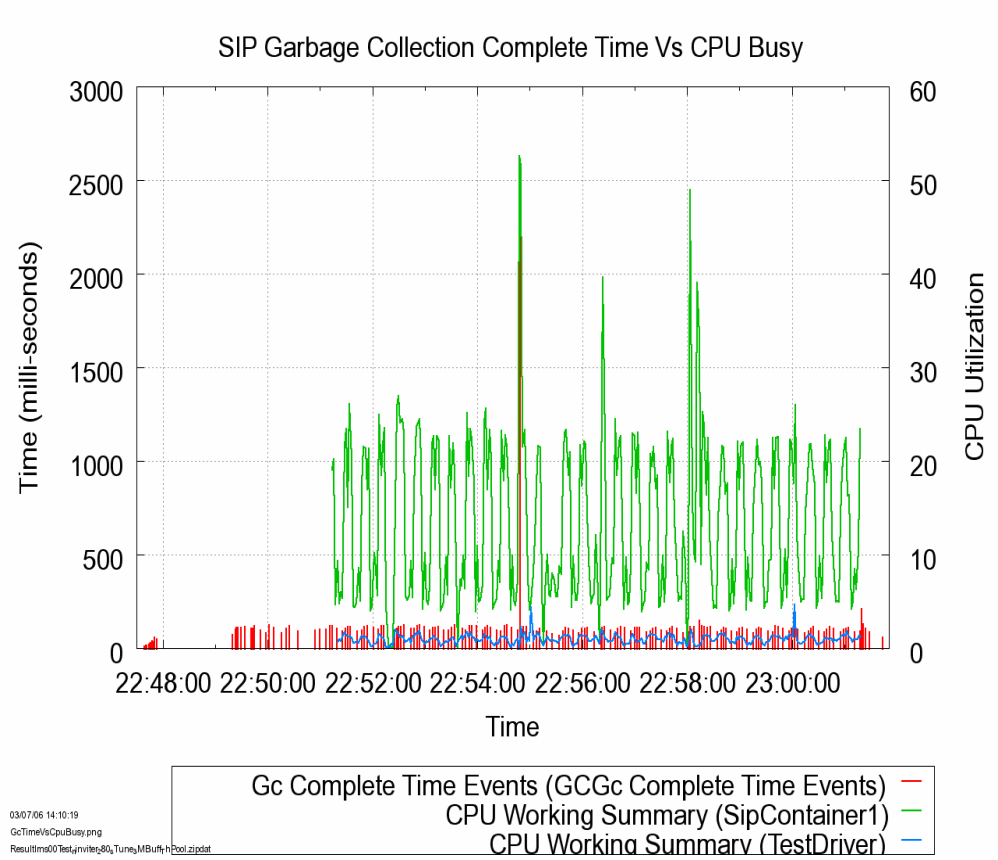

Figure 8: Effect of NSCD on Utilization

At some point in the stream of requests, either the name server considered this an incoming denial of service or the Linux TCP stack considered it to be a denial of service outgoing – the result was that all name lookups were ignored for a period of time. Once the nscd was started the problem disappeared.

The second analysis form is to generate statistics for the various factors during the steady-state portion of the measurements. The statistics that were recorded for each factor are: mean, standard deviation, 95th percentile, 99th percentile, maximum, minimum, sum of squares, and a count of the number of events. These statistics can then be compared for multiple runs, different configurations, etc. The statistics can then be used to calculate a transactional cost, such as the CPU cost per SIP dialog, CPU cost per SIP message, number of I/Os per SIP message, etc.

3 Conclusion

A tutorial on the performance of SIP, as it relates to an application server, has been reviewed. A high-level performance engineering methodology has been presented which is based on two years of experience in this area.

SIP can be thought of as a protocol with soft timing deadlines. For this reason, SIP differs markedly from HTTP. In the very near future, applications will converge so that they will receive input from both HTTP and SIP sources. This will present a continuing performance challenge which will be very interesting.

4. Acknowledgements

Many thanks are extended to Dazhi Wang, Mohit Jain, Jay Tunkel, and Don Boulia..

5. References

[1] RFC 3261

[2] RFC 1889

[3] JSR 116 - SIP Servlet Specification

[4] Erik Burckart. "Session Initiation Protocol in WebSphere Application Server V6.1 -- Part 1", IBM WebSphere Developer Technical Journal. http://www-128.ibm.com/developerworks/websphere/techjournal/0606_burckart/0606_burckart.html. June 2006

[5] IBM Developer Kit and Runtime Environment Java 2 Technology Edition, Version 5.0: Diagnostics Guide. http://www-128.ibm.com/developerworks/java/jdk/diagnosis/. Oct 2005.

[6] "java - the Java application launcher" http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html.

[7] Nigel Griffiths. “nmon performance: A free tool to analyze AIX and Linux performance”. http://www-128.ibm.com/developerworks/eserver/articles/analyze_aix/. Nov 2003.

[8] “ethereal - Interactively dump and analyze network traffic”. http://www.ethereal.com/docs/man-pages/ethereal.1.html.