Performance Measurements and Modeling of a Java-based Session Initiation Protocol (SIP) Application Server

Greg Franks
Department of Systems and Computer Engineering,
Carleton University
Ottawa, ON Canada K1S 5B6
greg,@sce.carleton.ca

Danny Lau
Department of Systems and Computer Engineering,
Carleton University
Ottawa, ON Canada K1S 5B6
hokching@gmail.com

Curtis Hrischuk
WebSphere Performance, IBM Software Group
Durham, NC USA 27703
cehrisch@us.ibm.com

ABSTRACT

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. Performance is a chief concern with SIP because Quality of Service is important and SIP has internal timers that need to be honored or network efficiency suffers. The Java community has even provided a standardized API so that SIP applications can now be built using Java 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 Java environment. In this paper, a Java 2 Enterprise Edition (J2EE) SIP application server's performance is analyzed in a standalone and cluster environment, with network traces used to build a performance model of each environment. This included gathering data from test runs and extracting performance parameters from packet traces to construct the performance models. The models are then calibrated to match the model prediction with real system test data. Using the calibrated models, some bottlenecks were identified and suggestions to improve the overall maximum throughput were developed and were subsequently implemented in the system.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies

General Terms

Performance

Keywords

Performance Analysis, Session Initiation Protocol, Ethernet, Two-phase Server, Layered Queueing Network

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) and 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 point, rather than using centralized control. This makes it extensible, scalable, and useful for mobile applications. SIP is becoming ubiquitous. Recently, application servers have been developed which allow both the HTTP and SIP protocols for the same application. It is expected that the next generation of web based applications will be of this fashion.

The Session Initiation Protocol (SIP) [1] is a signaling protocol designed to support a super set of the call-processing functions in the public switched telephone network (PSTN), but within an Internet Protocol communications network. These functions include the set up, modification, and tear down of voice and video calls over the Internet. Once a session is established, other protocols such as the Real-time Transport Protocol (RTP) [2] are used to carry the actual content.

SIP is a peer-to-peer protocol with much of the intelligence at the edge of the network (in contrast with Signaling System #7, the dominant PSTN protocol, which has dumb terminals and an intelligent core). However, for practical public use, SIP requires *proxy* and *registrar* network elements. Typically, an end-point or *"user agent"* will send a request to an intermediate proxy server which will forward the request either to another proxy or to the destination user agent. For example, in Figure 1, *bob@domain1.com* establishes a connection to *alice@domain2.com* through the *domain1* and *domain2* proxies, then communicates directly until Alice ends the connection.

The registrar is not shown in the figure but its purpose is quite intuitive. The registrar is simply the agent with which all the SIP users (called User Agents or UA) register to make themselves known when they first come on line. The UA's will also periodically re-register to indicate that they are still available. A non-intuitive aspect of the registrar is that its background load due to re-registrations can easily dwarf the load of new UA's coming on line, or going off-line.

Telephony applications often have very strict performance requirements. A performance model of a SIP application server would be of tremendous use during all stages of its life cycle. For example, during the initial design, the model

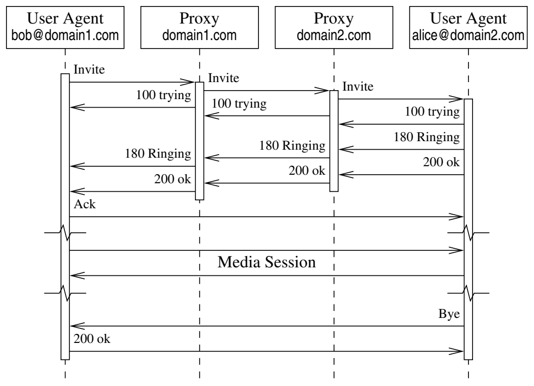


Figure 1: An example of a SIP exchange.

can be used to determine if the performance requirements for the application server can be met, locate potential problems prior to coding and to budget the demands of the various components of the project. During the implementation of a project, performance models can be used to locate performance problems in the design. Finally, once a product is constructed, performance models can be used for capacity planning so that a system of sufficient capacity is deployed. This is especially true for new technologies, where there is little guidance. For example, models could provide feedback on how to build a high-performance SIP Java application using the SIP Java application programming interface (called JSR 116 [3]).

Despite these benefits, performance models are often *not* used [4]. Part of the problem stems from the difficulty of constructing the model in the first place, moving from the software domain to the performance modeling domain. Constructing the model often requires the expert knowledge of a performance engineer to ensure that *just the right amount of detail* is included in the model. Furthermore, populating the model with service demands involves either estimation or measurement. End users are often skeptical of results based on estimates; using measurement is often labour intensive. Lastly, the marriage of SIP and Java is very new and there is very little guidance in the literature on how to construct performance models of this type, especially for a highly scalable architectural specification like Java 2 Enterprise Edition (J2EE).

The primary contribution of this paper is to present a method for constructing a performance through the analysis of Ethernet packets of a running application. The performance model chosen for this work is called a *Layered Queueing Network* (LQN) [5], which is particularly suited to distributed applications used today. The architecture of the performance model is derived from the behaviour of the system as defined by the Session Initiation Protocol (SIP) [1]. The parameters for the model are found using a Java implementation of SIP server by by running transactions (in this case they are SIP sessions) at low traffic rates; service times are then derived from the time stamps of the packets. The model is then solved and its results are validated against the live system under various load conditions.

The second contribution of this paper is a model of Ethernet behaviour using *two-phase* servers. The processors in today's hardware are becoming so fast that bottlenecks are now moving from their traditional sources of CPU's and disks and onto the communication channels between computers in the network. This work shows that simple queueing servers are not sufficient to model Ethernet connections accurately.

The remainder of the paper is organized as follows. First, the SIP scenarios used for the tests are described. Second, Layered Queueing Networks are briefly described as the performance model used here relies on the special *two-phase* servers which not commonly found elsewhere. Next, the actual model used to study the SIP system under test is described. Finally, results and conclusions, both for the system under test, and for the effect of the second phase are presented.

2. SIP TEST SCENARIOS

One of the most important SIP performance scenarios involves the Invite method, where two parties attempt to establish a connection with each other. This is the first scenario that occurs before two (or more) parties can do anything else. The end-to-end performance of this scenario is important because SIP's timer T1 has a 500 millisecond timeout period and will result in a message retransmit when it expires. Further, So, it is only appropriate to build a performance model of this scenario first.

Two machines are used for testing, each of which is a Blade Server running on Red Hat Enterprise Linux 3 Enterprise Server. Each Blade Server has multiple blades, and each blade is running an Intel dual-core hyper-threaded processor at 2.8 GHz. In both test scenarios, the load-generation "driver" software runs on one machine and the SIP server runs on the other. For the cluster configuration, the load balancer and the SIP Server run on a common chassis, but separate blades. The communication paths between blades on a Blade Server is at 4 gigabits per second and is on a switched Ethernet back plane. Communication between Blade Servers is through a 1 gigabit switch. Figure 2a shows the deployment diagrams for the two test cases described above.

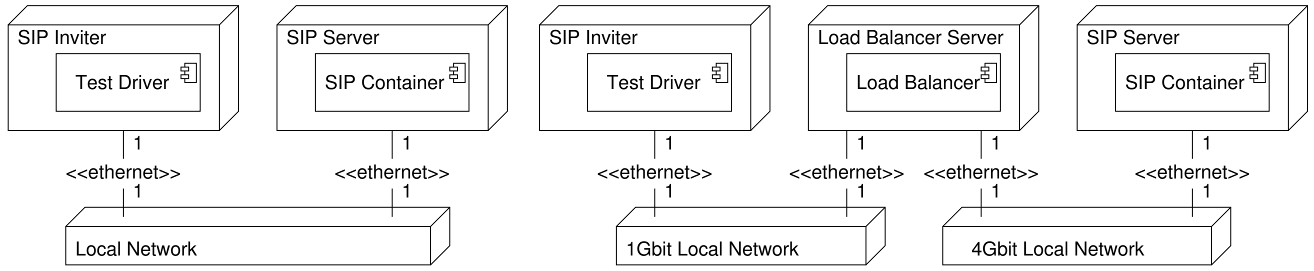
Figure 3 shows the sequence diagrams for the test cases use to construct a performance model. The scenario consists of an INVITE request, two redirection hops representing requests to proxy servers to establish a route, followed by an OK, indicating that the connection is established. Two different cases are shown:

(i) one where the SIP server interacts directly with the incoming request, and

(ii) another where a load balancer distributes the requests to multiple servers.

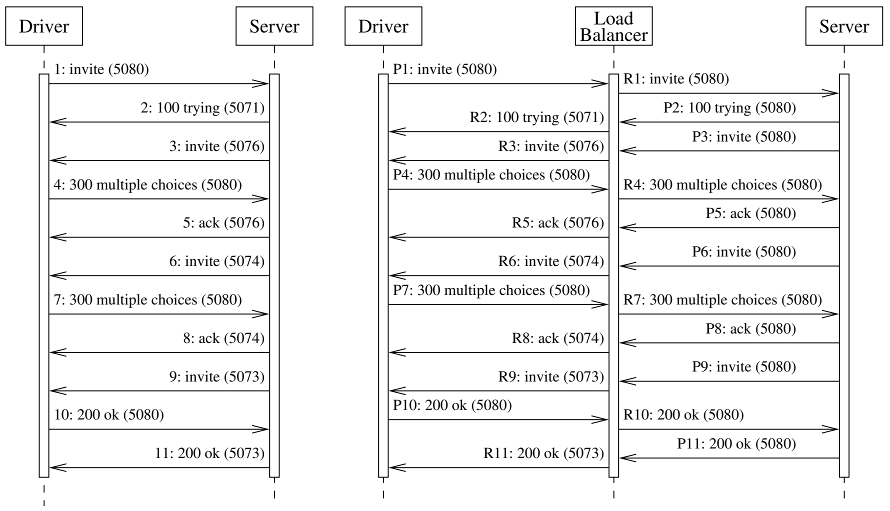
The SIP application server and application were written in Java. The application server was a J2EE server, such as described in [8]. The application used a Java application programming interface (API) similar to HTTP servlets [3]. This API provides an easy-to-use SIP programming model. In these tests, the application performed little business logic, executing only the minimum necessary functions for setting up a call as the objective of the performance testing was to test the SIP stack itself.

Measurements were conducted at one call per second, 50% and 100% server load, and with the server overloaded. However for this work, only the one connection per second and the 100% server load cases are considered. Wireshark [9] is



(a) Standalone (b) Cluster

Figure 2: Deployment diagrams for the two test environments



(a) Standalone (b) Cluster

Figure 3: Invite Scenario.

a packet sniffer that will record all the traffic on an Ethernet port, including fine grained timestamps, which is used in post-processing. Wireshark was run on each blade hosting the system under test to monitor the SIP message traffic. Traces from the one message per second traffic runs were used to derive the service times for the analytic models described below. The 100% server load case was found by increasing the offered traffic in the system until the point where message retransmissions began, indicating an overloaded system. These results were used to verify the results from the queueing model.

3. LAYERED QUEUEING NETWORKS

Layered Queueing Networks [5, 10, 11] are a form of extended queueing network model particularly suited for modeling systems with hierarchically nested resources which arise from using synchronous send-receive-reply interactions such as remote procedure calls. Layering arises from causality of requests from customers to servers in the system. These types of systems cannot be modeled directly using a conventional queueing network because the time a client is blocked on a server processing a request is a form of simultaneous resource possession. Since SIP interactions are of the request-reply type, a layered queueing network was used to model the system's performance. The next section briefly describes Layered Queueing Networks. The model used to study the SIP testbed performance follows.

3.1 Layered Queueing Networks

Figure 4 shows the major components of the Layered Queueing network meta-model. They are:

Processor: A *processor* is composed of a set of *tasks* and is used to execute the task's *phases*. They can represent actual CPU's, or can stand in to consume time for tasks representing customers or hardware devices. A processor may have more than one instance, which makes it a multi-server. A processor may also have an infinite number of instances, which makes it a delay server. In the models shown here (for example, Figure 8), processors are shown using a dashed box. The multiplicity of the processor is shown in the processor's label using the number in parenthesis.

Task: A task is used to model:

1. active entities such as a processes,
2. passive resources such as mutexes,
3. physical devices which are not CPU's such as disks, and
4. customers in the model (these tasks are called *reference tasks*).

Requests are queued at tasks in first-come, first-served order and are serviced by *entries*. Tasks are shown here using large parallelograms. A stacked task icon, such as *server* in Figure 8 denotes a multiserver

Entry: An *entry* is used to differentiate the types of service offered by a *task*. It invokes a sequence of one to three *phases* to perform the associated action. Entries are represented using the small parallelograms within task icons.

Phase: A *phase* is the lowest level of detail in the performance model. Phases execute in sequence on the *processor* associated with the task, and can make requests to *entries*. Entries which accept synchronous requests issue a reply after the first phase of execution completes; this behaviour is described in more detail below. The service time for a phase is specified as a list of values in square brackets for the corresponding entry. Phases are generalized into *activities*, described in [5].

Request: A *request*, with the exception of a forward, is made from a *phase* to an *entry*. A *Synchronous* request blocks the sender until the receiver replies. Synchronous requests can be *forwarded*, where the reply is treated as a synchronous request at another server which then either replies to the original client, or forwards the request to yet another server. *Asynchronous* requests do not block and cannot be forwarded.

Synchronous requests are shown using solid lines with filled arrow heads, asynchronous requests are shown with open arrow heads, and forwarded requests are shown using a broken line with a solid arrow head. Request rates are shown using labels in parenthesis associated with the arcs. If the source entry has multiple phases, then the request rates are also given as a list with each item in the list corresponding to a phase in the source.

The model supports many other features which are not described here. Most of these additional features are described in [5].

3.1.1 Two-phase Servers

A *two-phase* server is a server that, after replying to a request from a customer, continues to execute, shown in Figure 5. This type of server is a GNENP (General Non-Exhaustive Service, Non-Preemptive), SV (Single Vacation) server and has been studied previously in [12, §2]. Two-phase servers are often used to enhance performance because the client and server can run in parallel once the server replies [13]. For example, in Figure 5 the reply to the client is issued after phase-1; both the client and server can then run in parallel. However, this effect can also be used to *limit* performance because the client's subsequent request to the server cannot be serviced until its previous request has completed. This effect is called *overtaking*; the blocking time is shown as "`{t}`" in Figure 5b.

3.2 SIP Layered Queueing Model

The sequence diagrams in Figure 3 are used to construct the layered queueing network for the SIP system. The first three messages in this scenario are used to describe the basic

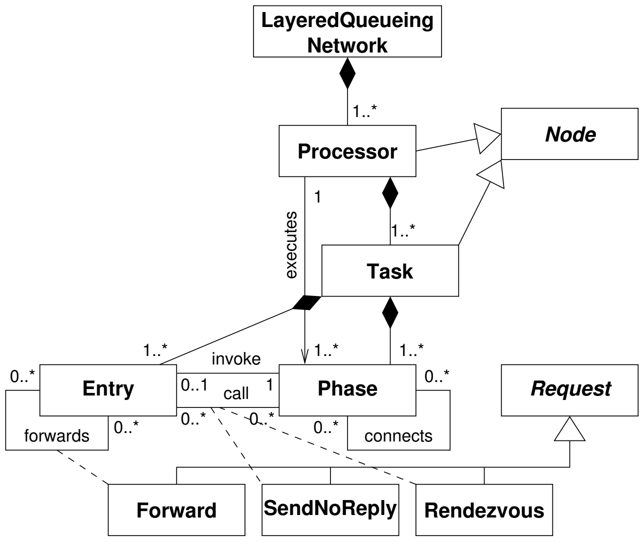
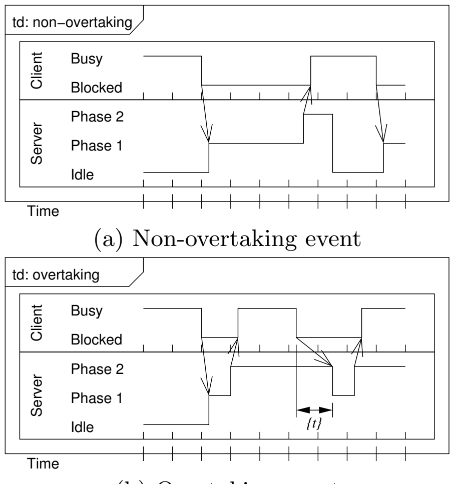


Figure 4: Layered Queueing Network Meta-Model.



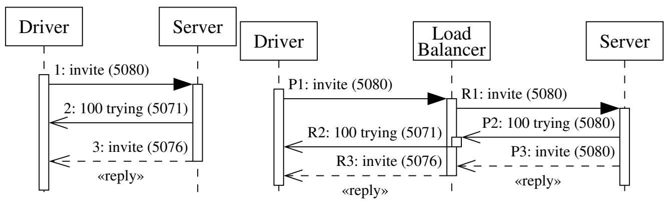
(a) Non-overtaking event

(b) Overtaking event

Figure 5: Two-Phase server time lines

template for the model. First, the SIP protocol uses *asynchronous* messages between parties [3]. However, the actual interactions are *synchronous* because Driver sends "`1: Invite`" to server which eventually responds with "`3: Invite`". This third message is actually a *reply* to the original invite request, so messages `1` and `3` can be converted into a synchronous interaction between the Driver and Server. Message `2` is an acknowledgment to message `1`, and is used to reset an internal timer used by the SIP protocol. This message can be modeled as an asynchronous request back to the Driver. Figure 6 shows the modifications to the sequence diagrams in Figure 3 for these three messages. The other interactions in the complete scenario are translated in a similar fashion.

Service times for the model were found by taking the difference in timestamps from a packet trace generated using Wireshark [9]. On modern Linux systems, these timestamps have microsecond precision because they are based on counting CPU cycles using the TSC register on Intel processors [14]. The system was run at a request rate of one call



(a) Standalone (b) Cluster

Figure 6: Modified `Invite` scenario using synchronous interactions.

per second to minimize the effects of queueing within the application server and the driver and to minimize any effects caused by the packet capture mechanism. After the run was completed, the packet traces from each processor were saved in separate files. Each trace file was then sorted by the SIP session ID, to separate each unique request from the driver, and by the time stamp. The result from this sort was a time-ordered sequence of messages for each SIP request, an example of which is shown in the trace summary in Figure 7. Service time parameters were then found by finding the average value of the differences in the time stamps of between sets of messages for each call flow.

The sections that follow describe the construction of the layered queueing model for the two scenarios in detail.

3.2.1 Standalone Model

Figure 8 shows the Layered Queueing Network model of the standalone system. The model consists of three *processors*: Pdriver, SIPserver and Network corresponding to the elements in the deployment diagram shown in Figure 2a. The Driver and the SIPServer act as peers, which causes a cycle in the call graph from the asynchronous `100 trying` and `ack` messages. The cycle is broken by splitting the driver into two tasks labeled driver and driverports respectively with driverports used to handle the asynchronous requests. The SIPServer task is modeled with ten threads which is the number of active threads configured in the system under test. The SIP application server process appears to run on only one processor so the SIPServer's processor is single-threaded. Finally, eleven requests are made from User to net. Each of these requests represents the delay incurred by each packet send from the driver to the SIP server and vice versa.

Table 1b lists the elapsed time (in milliseconds) between messages for the standalone system as measured by running Wireshark on the server's node. This approach takes into account the Network Interface Card and operating system overhead, as well as the application server cost. It does assume that the application server is the only process running on the system. Columns 1 and 2 identify the *to* and *from* messages in the sequence diagram in Figure 3a used to compute the time difference. From the server's standpoint, the arrival of the *from* message corresponding to the receive event, and the *to* message corresponding to the reply event. Since the server being driven at a very low traffic rate, it is assumed that the SIP server processing starts immediately upon reception of the message and continues until the reply is sent. Columns 3 and 4 list the mean service time and the 95% confidence interval after measuring three complete requests from the driver. Finally, column 5 lists the name entry in the LQN model in Figure 8.

Similarly, the data in Table 1a is used to find the service times for the client. The test driver is busy acting as a server from message 3 to 4, 6 to 7 and 9 to 10; the sum of these differences is used to populate the entry User of the driver task in Figure 8. However, the driver also receives messages 2, 5, and 8 (the acknowledgments) without sending replies. The upper bound on the service time for these cases is the time between the receipt of the acknowledgment and the receipt of the following invite and is used to set the service times for entries port5071 through port5076. Provided that the driver is not fully utilized, thus limiting the overall throughput of the system, errors here can be ignored.

Next, the data in Table 1c is used to find the Ethernet service time. Columns 1 and 2 again refer to the message number in the sequence diagram in Figure 3a. Columns 3 and 4 list the time difference between the *from* and *to* messages from the measurements taken from the Driver's node, and columns 5 and 6 list the time differences for the same two messages from the standpoint of the Server's node. The difference between column 3 and column 5 represents the transit time for two packets. The mean service time for one packet is one half of the mean of the differences for items listed in the table. This results in a measured delay of 0.0648ms with a 95% confidence interval of ± 0.0425.

Finally, the Ethernet has a finite capacity which is often much less than the advertised bandwidth. For the purposes of this model, the maximum utilization, η, was assumed to be 40%, which is the value measured by [15] for a switched Gigabit Ethernet and reported by Cisco [16] for the switch used by the system under test. To model this effect, the Ethernet model element, labeled as the net entry in Figure 8, is a phased. The phase one service time, s_1, is the value measured above. The phase two service time, $s_2 = 0.0972$mS, represents the unused 60% of the capacity, and is calculated using

$$s_2 = s_1 \left(\frac{1-\eta}{\eta} \right) \tag{1}$$

3.2.2 Cluster Model

Figure 9 shows the LQN model of the cluster system. This model adds an additional node, LoadBalancer, which is used to forward messages between the Driver and the SIPServer. Driver and SIPServer act as peers, as above, with LoadBalancer acting as an intermediary. The LoadBalancer was split into two proxies, balancer_up and balancer_down, to handle the synchronous and asynchronous requests respectively, in order to make the call graph acyclic. Further, the Load Balancer in the test setup was configured with ten threads, so this was modeled using a third task, labeled balancer in Figure 9, which executes the request on behalf of either balancer_up or balancer_down. This model also doubles the number of calls to the network task, because two messages are sent for every one message in the standalone model.

The service times between the driver and load balancer, and the load balancer and the server are different because of different network connections. They are derived using the data in Table 2 and the sequence diagram shown in Figure 3b. The service times for the SIP server in Figure 9 were calculated using the values in Table 2b for the messages R1 to P3, R4 to P6, R7 to P9 and R10 to P11. The values for the driver task were found in similar fashion using the data in Table 2a. The service times for the load balancer task was found using the data in Table 2c. When the driver is

```
 No  Session ID     Time      Diff      Src  Dst  SIP Message
  1  sip:3401@drv  4.946427  0.00071   drv  srv  Request:  INVITE sip:3401@drv
  2  sip:3401@drv  4.947137  0.001013  srv  drv  Status:  100 Trying
  3  sip:3401@drv  4.948150  0.000113  srv  drv  Request:  INVITE sip:3401@drv:5076
  4  sip:3401@drv  4.948263  0.000427  drv  srv  Status:  300 Multiple choices
  5  sip:3401@drv  4.948690  0.000204  srv  drv  Request:  ACK sip:3401@drv:5076
  6  sip:3401@drv  4.948894  9.2e-05   srv  drv  Request:  INVITE sip:3401@drv:5074
  7  sip:3401@drv  4.948986  0.000325  drv  srv  Status:  300 Multiple choices
  8  sip:3401@drv  4.949311  9.9e-05   srv  drv  Request:  ACK sip:3401@drv:5074
  9  sip:3401@drv  4.949410  0.000104  srv  drv  Request:  INVITE sip:3401@drv:5073
 10  sip:3401@drv  4.949514  0.000558  drv  srv  Status:  200 OK
 11  sip:3401@drv  4.950072  0.995326  srv  drv  Status:  200 OK
```

Figure 7: Wireshark Trace Summary of a Complete Call Flow. The message numbers in this figure correspond to the numbers used in Figure 3a.

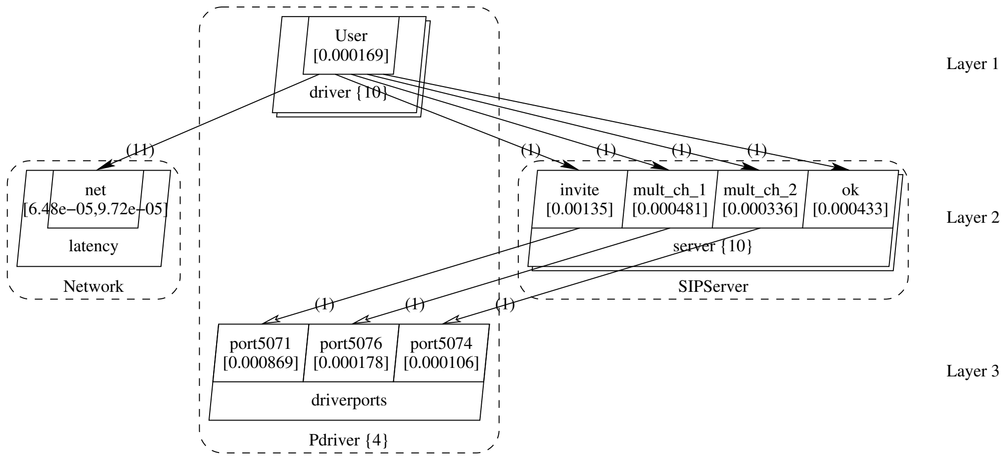


Figure 8: Layered Queueing Network of the Standalone configuration in §3.2.1.

making a synchronous request, the balancer task is invoked for both the request and reply, so the service times is the sum of the time the balancer handles each message. For example, the initial invite from the driver to the server consists of messages P1, R1, P3 and R3. The sum of the difference of P1 to R1 and P3 to R3 is the service time for entry *c1* on the balancer task. The asynchronous messages only involve one message transfer through the load balancer, for example messages P2 and R2, so the service times for entries processing these requests is simply the difference between the two messages.

The last consideration for this model is the Ethernet. Communication between the Driver and the load balancer is over a 1 gigabit Ethernet link with an estimated maximum available utilization of 40%. Communication between the load balancer and the SIP service is over a 4 gigabit channel with a measured maximum available utilization of about 20% [17] before SIP messages were dropped. Tables 2d and 2e list the time differences used to derive the Ethernet service time. From this data, the average service time for packets between the driver and the load balancer is 0.0616ms with a 95% confidence interval of ± 0.0029. Similarly, the average service time for packets between the load balancer and the SIP server is 0.0381ms with a 95% confidence interval of ± 0.0006.

4. RESULTS AND ANALYSIS

The models in the preceding section were validated by comparing the maximum predicted throughput against the maximum traffic load that the live system would support. The results of the comparison are shown in Table 3. For the standalone system, the model's throughput prediction is 5% higher than the measured throughput, and for the cluster system, the prediction is 4% lower. These results are quite good given that there is some uncertainty in the observed maximum throughput as this result was found by increasing the rate of requests from the test driver until timeouts were observed caused by lost messages. Table 3 also lists the utilizations for the SIP server and load balancer for both the live system and the model. The utilizations reported in this table are the overall utilization for the four effective processors in each node. Here, the model's predictions are consistently lower. This result is not surprising because the nodes are running other tasks, which are not captured by the performance model. It has also been experimentally verified that, in a moderately loaded system, the network can drop messages which will result in the additional load of the application server having SIP timers go off and a message retransmission (e.g., `200 OK` is dropped, causing the application server to retransmit an `invite`). Also, there will be more operating system overhead in the application server as the load increases due (e.g., context switching).

The bottleneck is a layered system is the entity found at the deepest level of the call graph with the highest uti-

Table 1: Standalone Test Environment Inter-Message Times (mS). Message numbers correspond to those used in Figure 3a.

(a) Driver

Message		Measured Time		Entry
From	To	$\overline{\Delta_t}$	$\pm 95\%$	
2	3	0.869	0.310	port5071
3	4	0.060	0.113	User
5	6	0.178	0.109	port5076
6	7	0.056	0.078	User
8	9	0.106	0.073	port5074
9	10	0.053	0.110	User

(b) Server

Message		Measured Time		Entry
From	To	$\overline{\Delta_t}$	$\pm 95\%$	
1	3	1.345	0.297	call1
4	6	0.481	0.074	call2
7	9	0.336	0.064	call3
10	11	0.433	0.308	call4

(c) Driver to Server

Message		Measured Time			
From	To	Driver		Server	
		$\overline{\Delta_t}$	$\pm 95\%$	$\overline{\Delta_t}$	$\pm 95\%$
1	3	1.458	0.570	1.345	0.297
4	6	0.580	0.137	0.481	0.074
7	9	0.432	0.114	0.335	0.064
10	11	0.542	0.043	0.333	0.308

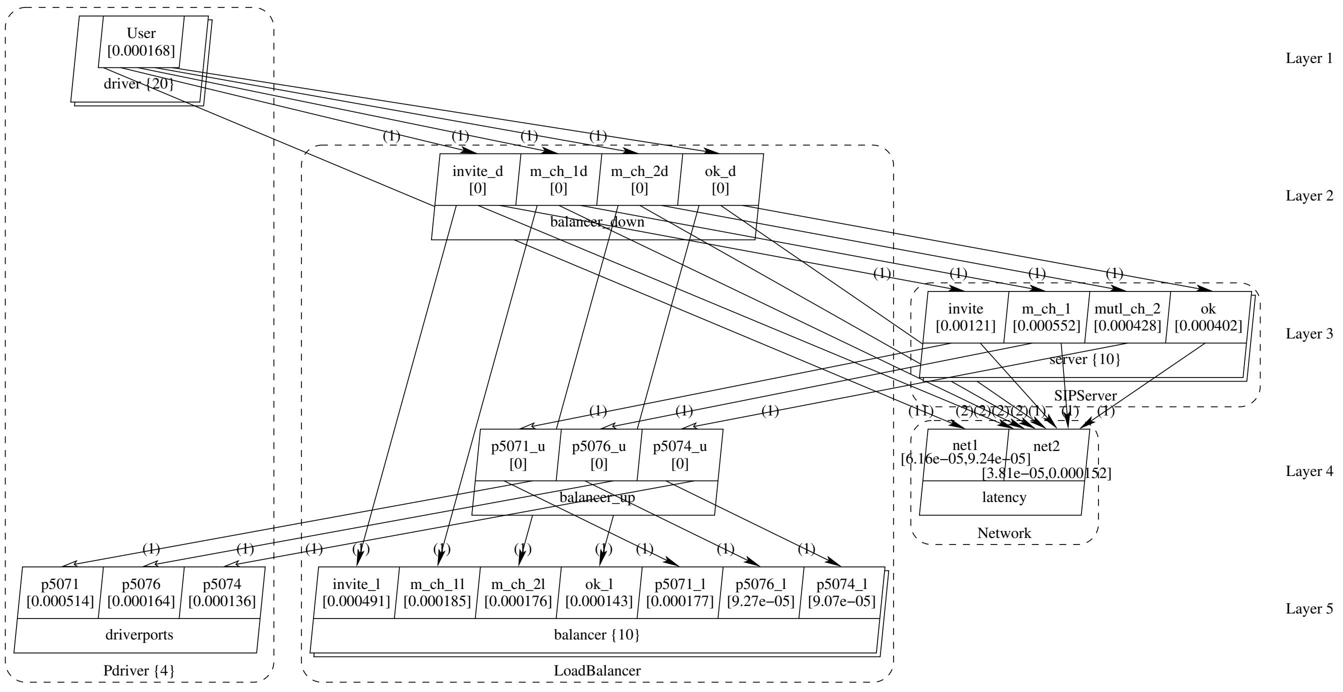


Figure 9: Layered Queueing Network of the Cluster configuration in §3.2.2.

Table 2: Cluster Test Environment Inter-Message Time (mS)

(a) Driver

Message		Measured Time		Entry
		Δ_t	±95%	
R2	R3	0.514	0.417	port5071
R3	P4	0.062	0.081	User
R5	R6	0.164	0.070	port5076
R6	P7	0.062	0.093	User
R8	P9	0.136	0.029	port5074
R9	P10	0.044	0.071	User

(b) Server

Message		Measured Time		Entry
		Δ_t	±95%	
R1	P3	1.211	0.622	server1
R4	P6	0.552	0.068	server2
R7	P9	0.428	0.056	server3
R10	P11	0.402	0.077	server4

(c) Load Balancer Service Times

Message				Measured Time		Entry
From	To	From	To	Δ_t	±95%	
P1	R1	P3	R3	0.491	0.163	c1
P2	R2			0.177	0.020	r1
P4	R4	P5	R5	0.185	0.035	c2
P5	R5			0.093	0.010	r2
P7	R7	P9	R9	0.176	0.004	c3
P8	R8			0.091	0.015	r3
P10	R10	P11	R11	0.143	0.009	c4

(d) Driver to Load Balancer

Message		Measured Time			
From	To	Driver		Load Balancer	
		Δ_t	±95%	Δ_t	±95%
R1	P3	1.290	0.625	1.212	0.623
R4	P6	0.629	0.071	0.552	0.068
R7	P9	0.506	0.055	0.428	0.056
R10	P11	0.476	0.077	0.402	0.077

(e) Load Balancer to Server

Message		Measured Time			
From	To	Load Balancer		Server	
		Δ_t	±95%	Δ_t	±95%
P1	R3	1.899	0.788	1.780	0.785
P4	R6	0.946	0.067	0.814	0.070
P7	R9	0.812	0.060	0.681	0.057
P10	R11	0.729	0.072	0.618	0.072

Table 3: Results

	Standalone			Cluster		
Result	Meas.	Pred.	Err.	Meas.	Pred.	Err.
Throughput	350	369	5%	290	280	-3%
Server Utilization	31%	24%	-23%	28%	18%	-36%
Load Balancer Utilization	Not Applicable			10%	9%	-10%

lization [18]. If the entity is a leaf node in the call graph, the bottleneck is a device, otherwise it is a task. Table 4 show the utilization results for each of the entities (except the driver) for the two configurations. For the standalone configuration, the bottleneck is the SIP Server itself. This bottleneck can be mitigated by either improving the performance of the software, or by spreading the load to more processors. For the cluster model, the network is the bottleneck.

Entity	Standalone	Cluster
net	0.66	**1.00**
balancer task		0.47
Balancer proc.		0.32
server task	2.12	4.57
SIPServer proc.	**0.96**	0.73

Table 4: Utilization Results

4.1 Phased Ethernet Model

One of the novel aspects of this performance model is the use of a two-phase server to model the Ethernet. In this work the second phase is used as a vacation in order to limit throughput (second phases are usually used to improve performance by increasing parallelism between the client and the server [13]). For the cluster model, the maximum utilization of the 1 gigabit external Ethernet was assumed to be 40%, and the maximum utilization of the 4 gigabit internal Ethernet was measured to be approximately 20%. Further, the service time measurements on the live system had some spread. To explore the effects of these variations, a factorial experiment was run by varying the maximum Ethernet utilization, η, by ±10% and by varying the measured Ethernet service time, s_1, from its lower to upper confidence interval limit.

Figure 10 shows the results from varying the service time of the two Ethernets as a surface plot. The high and low plots represent the upper and lower bounds of the throughput using the confidence intervals for the Ethernet service time. Figure 10 also shows that when the maximum utilization of the Ethernet improves to 50% on the 1 gigabit link, and 30% on the 4 gigabit link, a new bottleneck (the SIP server) takes over to limit throughput.

Figure 10 also shows the 290 calls per second throughput contour – the measured value from the live system. This contour gives the feasible values for the maximum available utilizations for both Ethernet connections.

4.2 Sources of error

Two simplifying assumptions were made that can lead to errors in the results from the analytic model. First, it was assumed that the SIP application running on the server made no requests to lower level servers, such as disks, not also running on the server. If the SIP server were to do so, the blocking time on the lower-level device would be reflected in the measured service time, and therefore the predicted utilization of the SIP server's CPU. If the CPU is a bottleneck, the predicted throughput would be lower than observed. However, the results in Table 3 show that the predicted CPU utilization at server is lower than the measured value, so the assumption appears to be valid. Second, is was assumed that the SIP server ran to completion for each service request received during the measurement runs. Context switching would lengthen the service time and raise the predicted CPU utilization. Again, this assumptions appears to be valid.

Another source of error in the performance model arises from the asynchronous messages, for example, the *100 trying* from the server to the driver. No indication is given by the SIP Driver when the service for these message completes, so the service time at the driver is assumed to be the difference between the next request from the server and preceding asynchronous acknowledgment. This time interval will be too long, raising the predicted utilization at the driver's CPU. However, provided that this device is **not** a bottleneck in the model, this error can be ignored as it will not limit the predicted throughput.

Finally, the measurements only capture the load caused by SIP processing. The blades in the system are also running other daemons which consume CPU time. Further, the Java garbage collector runs periodically, stopping all service for external requests. This extra load is not captured in the model, so the predicted utilizations will be lower than the measured values. One mitigating factor, however, is that each blade is running a multiprocessor, so that the extra load caused by services other than those used for SIP processing can run in parallel with the SIP processing. The assumption used for the performance model is that the SIP server had the exclusive use of exactly one processor.

5. CONCLUSIONS

This paper has presented a layered queueing network performance model of a prototype JSR116 compliant SIP application server. Two systems were studied: a standalone system where clients interact directly with the SIP server, and a cluster system where a SIP load balancer is interposed between the two parties. The parameters for the performance model were found by using timestamps from packet traces from the various servers in the system. In both cases, the capacity predicted by the model was within 5% of the capacity measured from the prototype system.

The utility of the performance models here comes from locating bottlenecks in the performance of the system. For the standalone configuration, the performance bottleneck is the SIP server itself. This result is somewhat curious in that the measured utilization of the processor is only 31%, suggesting plenty of capacity to spare. However, this is the aggregate utilization of all of the processors in the Intel chip. The performance model implies that only one of the four processing threads is actually being used to process requests. This suggests that improvements are needed to the application to exploit parallelism on the processor. For the cluster configuration, the bottleneck is the Ethernet connection. The extra traffic put on the shared link is saturating the system. Unfortunately, it is not known whether this bottleneck is caused by the Ethernet hardware itself, or from a lack of resources, such as buffers, at one of the blades in the test system.

Finally, this paper has presented a novel way of modeling Ethernet by using a two-phase fixed-rate server. The first phase is used to represent the time the Ethernet is busy transmitting the packet, while the second phase is used to represent the time the Ethernet is "on vacation" and unable to accept new requests. Because of these vacations, the Ethernet can be saturated even though its maximum throughput is significantly lower than the advertised bandwidth.

Acknowledgments

The authors gratefully acknowledge the assistance of IBM for providing equipment for conducting this research and the financial support of NSERC (Natural Sciences and Research Council of Canada).

6. REFERENCES

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. J. J. Peterson, R. Sparks, M. Handley, E. Schooler, RFC 3261: SIP: Session initiation protocol (Jun. 2002).
URL `http://www.ietf.org/rfc/rfc3261.txt`

[2] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: A transport protocol for real-time applications (Jul. 2003).
URL `http://www.ietf.org/rfc/rfc3550.txt`

[3] A. Kristensen, SIP servlet API 1.0 specification, http://www.jcp.org/aboutJava/communityprocess/final/jsr116/ (feb 2003).

[4] M. Woodside, G. Franks, D. C. Petriu, The future of software performance engineering, in: L. C. Briand, A. L. Wolf (Eds.), Future of Software Engineering FOSE 07, Minneapolis, MN, USA, 2007, pp. 171–187. `doi:10.1109/FOSE.2007.32`.

[5] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, Enhanced modeling and solution of layered queueing networks, IEEE Trans. Softw. Eng. 35 (2) (2009) 148–161. `doi:10.1109/TSE.2008.74`.

[6] C. E. Hrischuk, C. M. Woodside, J. A. Rolia, R. Iversen, Trace-based load characterization for generating performance software models, IEEE Trans. Softw. Eng. 25 (1) (1999) 122–135. `doi:10.1109/32.748921`.

[7] T. Israr, M. Woodside, G. Franks, Interaction tree algorithms to extract effective architecture and layered performance models from traces, J. Syst. and Soft.

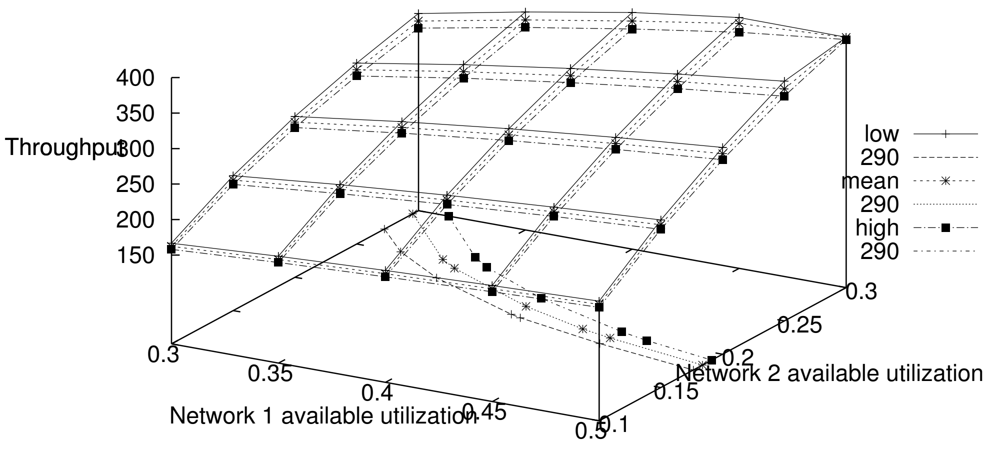


Figure 10: Throughput versus maximum Ethernet utilization using a two-phase server.

80 (4) (2007) 474–492. doi:10.1016/j.jss.2006.07.019.

[8] E. Bayeh, The WebSphere Application Server architecture and programming model, IBM Syst. J. 37 (3) (1998) 226–348. doi:10.1147/sj.373.0336.

[9] Wireshark: A network protocol analyzer, www.wireshark.org.

[10] J. A. Rolia, K. A. Sevcik, The method of layers, IEEE Trans. Softw. Eng. 21 (8) (1995) 689–700. doi:10.1109/32.403785.

[11] C. M. Woodside, J. E. Neilson, D. C. Petriu, S. Majumdar, The stochastic rendezvous network model for performance of synchronous client-server-like distributed software, IEEE Trans. Comput. 44 (8) (1995) 20–34. doi:10.1109/12.368012.

[12] B. Doshi, Single server queues with vacations, in: H. Takagi (Ed.), Stochastic Analysis of Computer and Communication Systems, North Holland, 1990, pp. 217–265.

[13] G. Franks, M. Woodside, Effectiveness of early replies in client-server systems, Performance Evaluation 36 (1) (1999) 165–184. doi:10.1016/S0166-5316(99)00034-6.

[14] A. Pásztor, D. Veitch, PC based precision timing without GPS, in: Proc. ACM SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems, Marina Del Rey, California, 2002, pp. 1–10. doi:10.1145/511334.511336.

[15] P. A. Farrell, O. Hong, Communication performance over a gigabit Ethernet network, in: Proc. IEEE International Performance, Computing, and Communications Conference, 2000. (IPCCC '00), Phoenix, AZ, USA, 2000, pp. 181–189. doi:10.1109/PCCC.2000.830317.

[16] C. Systems, Cisco 7500 gigabit ethernet interface processor (geip), White Paper (1998).

[17] C. Hrischuk, G. Deval, A tutorial on SIP application server performance and benchmarking, in: The 32nd International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems CMG 2006, Vol. 2, Computer Measurement Group, Reno, NV, 2006, pp. 729–740.

[18] G. Franks, D. Petriu, M. Woodside, J. Xu, P. Tregunno, Layered bottlenecks and their mitigation, in: Proc. 3rd International Conference on the Quantative Evaluation of Systems (QEST'06), Riverside, CA, USA, 2006, pp. 103 – 114. doi:10.1109/QEST.2006.23.