Evaluating SIP Proxy Server Performance

Erich M. Nahum, John Tracey, and Charles P. Wright
IBM T.J. Watson Research Center
Hawthorne, NY, 10532
{nahum,traceyj,cpwright}@us.ibm.com

ABSTRACT

SIP is a protocol of growing importance, with uses for VoIP, instant messaging, presence, and more. However, its performance is not well-studied or understood. In this paper we experimentally evaluate SIP proxy server performance using micro-benchmarks meant to capture common SIP proxy server scenarios. We use standard open-source SIP software such as OpenSER and SIPp, running on an IBM BladeCenter with Red Hat Enterprise Linux and Gigabit Ethernet connectivity.

We show performance varies greatly depending on how the protocol is used. Depending on the configuration, throughput can vary from hundreds to thousands of operations per second. For example, we observe that the choice of stateless vs. stateful proxying, using TCP rather than UDP, or including MD5-based authentication can each affect performance by a factor of 2–4.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Applications; C.5.5 [**Computer System Implementation**]: Servers; D.4.8 [**Performance**]: Measurements

General Terms

Measurment, Performance, Experimentation

Keywords

SIP, Servers, Performance, Experimental Evaluation

1. INTRODUCTION

The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, maintaining, and tearing down sessions for various types of media, including voice, video, and text. SIP is of growing importance, as it is being used for many media-oriented applications such as Voice over IP (VoIP), voicemail, instant messaging, presence, IPTV, network gaming, and more. It is also the core protocol for the IP Multimedia Subsystem (IMS), the basis for the 3rd-Generation Partnership Program (3GPP) for both fixed and wireless telephone networks. SIP relies on an infrastructure of servers, which are responsible for maintaining the locations of users and forwarding SIP messages across the application-layer SIP routing infrastructure toward their eventual destinations.

The performance of these SIP servers is thus crucial to the operation of the infrastructure, as they can have a primary impact on the latency of media applications, e.g., for initiating a phone call. However, SIP server performance is not well-studied or understood. Service providers clearly require performance information to understand how to provision their infrastructures to provide reasonable QoS.

The goal of this paper is to shed more light on how SIP proxy servers perform under various configurations and explain some of the limits to performance. We evaluate server throughput and latency for common SIP proxy server configurations, using micro-benchmarks on a dedicated experimental testbed. In particular, we are interested in identifying the primary factors that determine SIP proxy server performance including:

- Where is the time spent in servicing SIP requests?
- How significant are security costs such as authentication and encryption?
- How does the choice of stateless vs. stateful proxying affect performance?
- What is the impact of the transport protocol on performance?

We study these issues experimentally with standard open-source SIP software. We use a common SIP proxy server, Open SIP Express Router (a.k.a. OpenSER), running on an IBM BladeCenter with a 3.06 GHz Intel Xeon. The blade runs Red Hat Enterprise Linux 4 update 3, with a 2.6.17.8 kernel. Performance is measured by clients using the SIPp workload generator sending requests over a private copper Gigabit Ethernet. We measure throughput, average response time, and distributions of response times for a given load level, driving the system not only to capacity but into overload as well.

We find that SIP performance, in terms of throughput, can vary by an order of magnitude, depending on how the server is configured. Proxying throughput can vary from hundreds to thousands of operations per second, depending on whether authentication is used, whether transactions are

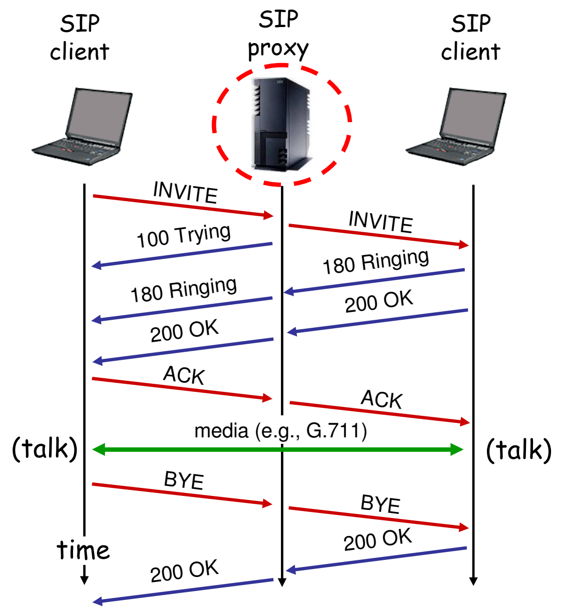

Figure 1: Stateful Proxying

stateful or stateless, and whether the underlying transport protocol is UDP or TCP. Authentication can reduce performance by a factor of 4, depending on the scenario. Using TCP as the transport can reduce performance by a factor of 3, and stateful configuration for proxying can cut performance in half. We show that the distributions of response time vary radically and degrade substantially when the system is under overload.

The result is that organizations deploying SIP technology must be aware of how their systems are configured and used, as this will have a primary influence on the performance of their systems, and thus how many resources need to be provisioned.

The rest of this paper is organized as follows: Section 2 presents the SIP proxy server configurations that we evaluate. Section 3 describes our experimental setup, and Section 4 presents our results in detail. Section 5 discusses some related work, and Section 6 provides our summary and conclusions and briefly presents plans for future work.

2. PROXYING SCENARIO

In this Section, we describe the common SIP proxy server scenario that we evaluate. Proxying is the core SIP function of forwarding a SIP message towards its eventual destination in the SIP infrastructure. In this section, we describe 4 potential scenarios: stateful vs. stateless proxying, both with and without authentication.

Figure 1 shows an example of stateful proxying without authentication. The hashed circle around the proxy illustrates that this is the component ("system under test") that we are measuring. In this example, the first SIP client wishes to establish a session with the second SIP client and sends an `INVITE` message to the proxy. Since the proxy is stateful, it responds with a `100 TRYING` message to inform the client that the message has been received and that it need not worry about hop-by-hop retransmissions. It then looks up the contact address for the SIP URI of the second client and, assuming it is available, forwards the message. The second client, in turn, acknowledges receipt of the message and informs the proxy that it is notifying the user via the `180 RINGING` message. The proxy then forwards that message to the initiator of the `INVITE`, informing the client that the end host has received the message and that the line is "ringing." The user on the second client machine then accepts the call, generating a `200 OK` message, which is sent to the proxy which forwards it on to the first client. The first client then generates an acknowledgment. Having established the session, the two endpoints communicate directly, peer-to-peer, using a media protocol such as RTP [10]. However, this media session does not traverse the proxy, by design. When the conversation is finished, the first user "hangs up" and generates a `BYE` message that the proxy forwards to the second user. The second user then responds with a `200 OK` which is forwarded back to the first user.

The above example is for a transaction-stateful, dialog stateful scenario where all SIP messages are routed through the proxy, using the `Record-Route:` header option. SIP proxies may be configured so that not all messages need traverse the proxy. For example, the `BYE/OK` exchange could be sent directly between the two clients. The above scenario is frequently used, however, since it enables per-call accounting and billing.

The other 3 proxying scenarios are straightforward extensions of the first.

In the stateless proxying scenario without authentication, the call flows are the same as in Figure 1, except that there is no `100 TRYING` message sent from the proxy to the client. In this case, the proxy does not create local state based on the transaction and relies on the endpoints to retransmit lost messages.

In the stateful proxying scenario with authentication, after receiving the initial `INVITE`, the proxy responds with a `407 UNAUTHORIZED` message, challenging the client to provide credentials that verify its claimed identity with a response based on that challenge. The client then retransmits the `INVITE` message with the generated credentials in the `Authorization:` header. The proxy also challenges `BYE` requests, requiring the UAC to retransmit the `BYE` with the proper `Authorization:` header, to prevent unauthorized hang-ups.

The stateless proxying scenario with authentication is similar to the previous scenario, except that there is no `100 TRYING` provided to the client and no transaction or dialog state created on the proxy.

Because SIP allows the use of multiple transport protocols, including UDP, TCP, SCTP, and SSL, we also wish to evaluate the impact of the choice of transport on performance. In our experiments, we evaluate UDP and TCP. In the case of UDP, all requests and responses are routed through a single connectionless UDP socket. With TCP, each client machine uses a separate persistent TCP connection to the registrar, proxy, or redirector, as appropriate to the scenario.

3. EXPERIMENTAL TESTBED

In this Section we describe the software and hardware utilized in our experiments.

3.1 SIP Server Software

We use the Open SIP Express Router version 1.1.0 (OpenSER) [12], a freely-available, open source SIP proxy server. OpenSER is a "fork" of SIP Express Router (SER) [3], sharing much of its code base. Both proxies are written in C, use standard process-based concurrency with shared memory segments for sharing state, and are considered to be highly efficient. Janak's thesis [4] describes many performance optimizations that are utilized in SER (and by implication, OpenSER). Each proxy has large feature sets, considerable user bases, active mailing lists, and third-party contributions (e.g., from `sip.edu` and `onsip.org`). We chose OpenSER over SER due its better documentation, but we believe our results will hold with SER as well.

In configurations where a user database was required, we use MySQL [11] 4.1.12-3.RHEL4.1, which we populated with 10,000 unique user names and passwords. OpenSER is configured to use a write-back caching policy, to maintain client state across restarts but also to achieve close to in-memory DB performance.

3.2 SIP Client Workload Generator

We use the SIPp [2] SIP workload generator, another freely available open-source tool. SIPp allows a wide range of SIP scenarios to be tested, such as user-agent clients (UAC), user-agent servers (UAS) and third-party call control (3PCC). SIPp is also extensible by writing third-party XML scripts that define new call flows; we wrote new flows that were not included with SIPp to handle authentication. SIPp has many run-time options we took advantage of, such as multiple transport (UDP/TCP/TLS) support; MD5-based hash digest authentication, and scriptable support to allow calls to be generated from a list of users.

We made several modifications to SIPp to improve its performance, so as to reduce the amount of client resources required to drive the server to saturation. Most importantly, we removed a restriction that limits the number of outstanding calls to three times the requested load level. This artificially limits the offered load, effectively making SIPp a *closed-loop* workload generator. Since we are interested not only in the maximum capacity of the server but also how well it behaves under overload, we removed this limitation in SIPp. Our improvements have been contributed back to the open source community and have been incorporated into the latest development releases of SIPp.

3.3 Client and Server OS Software

The servers in our experiments uses RedHat Enterprise Linux AS Release 4 update 3, using a locally-built Linux kernel 2.6.17.8, which is more recent than the 2.6.9 kernel variant that ships with RedHat. Our client machines use the SuSE SLES 9 release 2 Enterprise distribution, with a 2.6.5-9 kernel.

3.4 Hardware and Connectivity

The server hardware used is an IBM blade server residing in an IBM BladeCenter. The blade has 2 Intel Xeon 3.06 GHz processors with 4 GB RAM and 100 GB Toshiba MK4019GAXB ATA disk drives. However, for our experiments, we only use one processor. The blade has 2 Broadcom NetXtreme BCM5704S Copper Gigabit interfaces; each interface is connected to a separate Nortel Gigabit switch that is included with the BladeCenter. One switch is connected to our building's regular LAN, while the other is connected to our private experimental network. To minimize experimental perturbation and variability, all of our measurements are conducted over the experimental network, where minimal other traffic occurs (e.g., spanning-tree). Also residing on the private experimental network are 10 client machines used for load generation; half as UACs and half as UASs. Each client machine is an IBM Intellistation with a 1.7 GHz Intel Pentium 4 processor, 512 MB of RAM, an 18 GB SCSI disk, and an Intel E1000 Gigabit Ethernet adapter.

3.5 Experiments and Metrics

In our experiments, we wish to measure both throughput and latency as a function of load on the server. Throughput is relatively straightforward to define, in terms of the number of the appropriate completed operations per second.

Latency is defined as the time between when the `INVITE` is sent and the eventual successful `200 OK` is received. This is the latency as perceived by the user for initiating a call, which we believe is of more interest than latency that includes the call duration or termination (i.e., `BYE`). This is similar to the *Session Request Delay (SRD)* as defined in [5], except that there the latency timer is stopped when a `180 RINGING` response is received. Since there is no programmed delay between the `180 RINGING` and the `BYE` on the UAS, we believe the difference is minimal.

For each metric (throughput, latency, and CPU profile) that we report, the number is the average over 5 runs. Latency and throughput curves include 95th percentile confidence intervals. Each run lasts for 120 seconds after a 5 second warm-up time. We also show the cumulative distribution function (CDF) of response times for various load levels, to illustrate how response time varies with load, particularly at 95th and higher percentiles.

3.6 Restrictions, Limitations, and Scope

Note that our setup by no means covers the entire space of configurations for SIP. Results for two major SIP server scenarios, registration and redirection, are excluded due to space limitations, but are available in a companion technical report [6]. We do not consider non-VoIP scenarios such as Instant Messaging or Presence. In addition, there are many VoIP situations not measured by our experiments, including outbound proxying, PSTN gatewaying, ENUM processing, SSL and SCTP as a transport layer, or error processing for unregistered or unauthenticated users. Each of these presents opportunities for future work.

4. RESULTS

Before detailing our proxying results, we believe it is necessary to mention a significant performance fix for transaction-stateful proxying that influences many of our results.

While examining the CPU profiling results generated using oprofile, one problem we observed very quickly was the extremely large amounts of CPU cycles spent in the OpenSER module responsible for transaction-stateful processing. Figure 2 shows an example of this, in the left-hand bar marked "Original."

Looking more closely, we saw that this time was coming from a single function, `insert_timer_unsafe()`, which inserted new transactions into a timer structure for retransmissions in the future. This list is sorted by expiration time, yet the routine needlessly searched through the list even

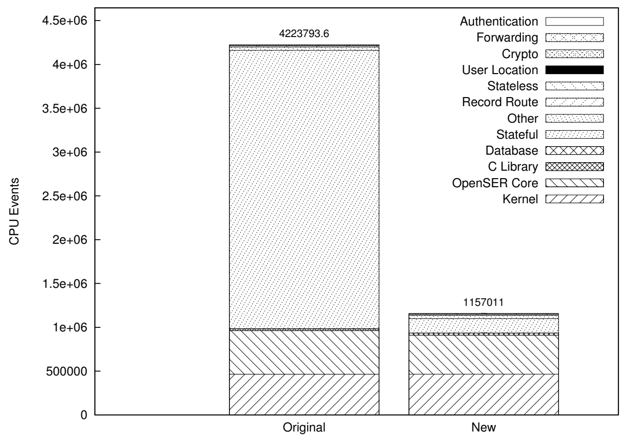

Figure 2: CPU Profiles: Transaction-Stateful Proxying, UDP, No Authentication, Load 1000 CPS

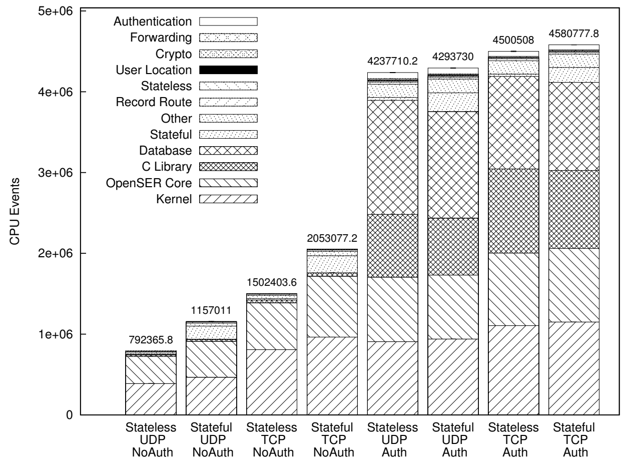

Figure 4: CPU Profiles: Proxying

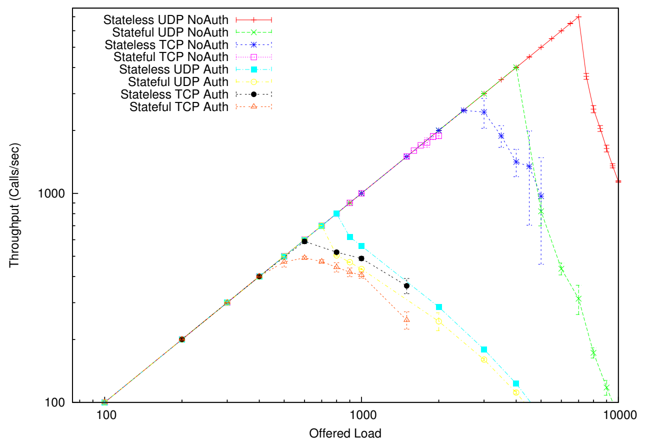

Figure 3: Throughputs vs. Offered Load: Proxying

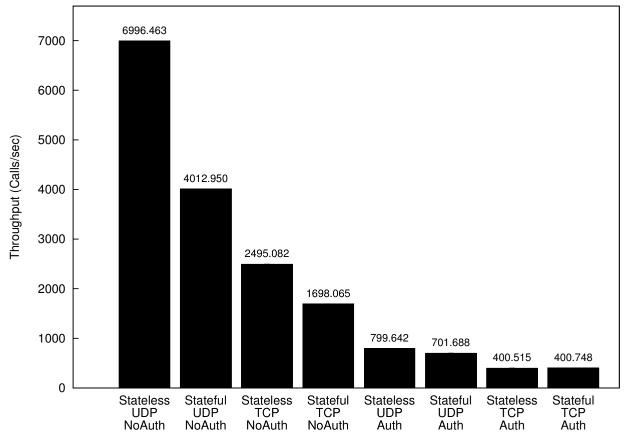

Figure 5: Peak Throughputs: Proxying

though the timer needed only to be appended to the end of the list. This function becomes a bottleneck because, at high loads, each new call results in two new transactions (the `INVITE` and the `BYE`), each of which requires a timer to be set and canceled in the common case. This bug is also present in the released version (0.9.6) of SER.

A one-line fix corrected this problem, increasing the peak throughput by over a factor of ten in the transaction-stateful scenario without authentication, from 400 to 4012 calls per second. Performance increases by 250 percent for the scenario with authentication, from 300 to 701. Figure 2 also shows the CPU profile for the same load after applying our fix, in the right-hand bar marked "New." All results reported below include our fix.

Throughputs

Figure 3 shows throughputs versus offered load for stateful and stateless proxying, with and without authentication, using both UDP and TCP as transport protocols. Note that both X and Y axes are in log scale. Peak throughputs for each curve are also reported in Figure 5. Peak throughputs are calculated as the maximum throughput achieved while maintaining at least a 99 percent success rate.

As can be seen in Figures 3 and 5, the achieved throughputs vary considerably, depending on on how the systems are configured. Starting with the results for stateless proxying with UDP and no authentication as a "best case," we can illustrate how the various features and functions influence performance.

The most significant feature that influences performance is whether authentication is used. Depending on the configuration, enabling authentication can reduce performance anywhere from 60 percent (in the stateful TCP case) to 90 percent (in the stateless UDP case). CPU profiles for these tests, given in Figure 4, illustrates why performance degrades with authentication. Observe that when authentication is enabled, the profiles show almost half the cycles are spent in the MySQL database and the standard C library functions. Neither of these components are significant when authentication is not enabled; thus, we attribute the C library usage to MySQL. The actual MD5 hash calculation, shown in the profile under the 'Crypto' heading, is typically less than 1 percent. The reason is that the database

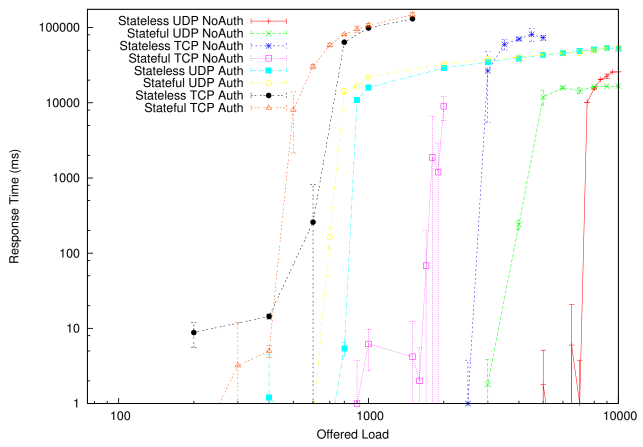

Figure 6: Average Response Time: Proxying

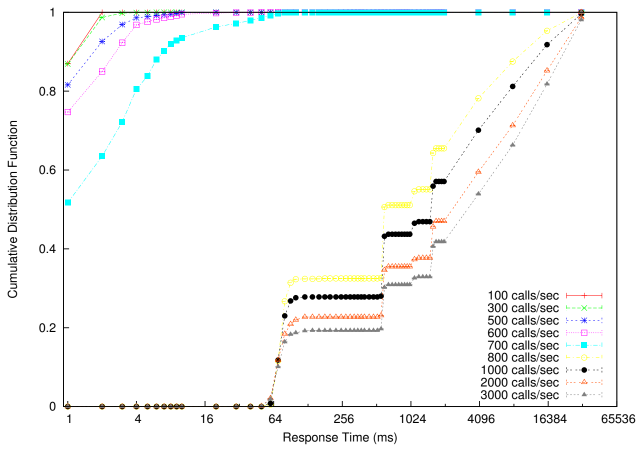

Figure 7: Response Time CDF: Stateful Proxying, UDP, with Authentication

is consulted much more aggressively when authentication is used, even though OpenSER is configured to use a write-back caching policy, as described in Section 3.1. A straightforward solution would, of course, be to locate the DB on a separate machine, but that would be increasing the resources available, and we wish to study the performance limits of a single node in this work.

The next most significant performance feature is which transport protocol is used, TCP or UDP. Using TCP can reduce performance anywhere from 43 percent (the stateful proxying scenario with authentication) to 65 percent (stateless proxying without authentication). Looking at Figure 4, one can see that the time spent in the OpenSER core goes up significantly, and that the time spent in the kernel almost doubles. TCP is a much more complex protocol than UDP, providing much more functionality, and thus requires significantly larger code paths.

Finally, we see that the choice of stateless vs. stateful processing can also have a significant impact on performance, depending on the configuration. Enabling stateful processing can reduce performance by as much as 60 percent (for the proxying configuration using TCP with no authentication) to having effectively no impact on performance (in the configuration using TCP with authentication).

Observe also that OpenSER does not preserve throughput under overload, as achieved throughput falls quickly when load exceeds the capacity for that configuration. Ideally, a system should maintain maximum throughput even when subjected to overload; this is difficult to achieve in practice, of course, and is the subject of active research. This demonstrates that overload management and control are issues in OpenSER for the future.

Latencies

Figure 6 shows average response times versus load. Note that both X and Y axes use log scales. SIPp has a 1 millisecond timer granularity; thus, any responses that occur within less than a millisecond are treated as zero. Thus, many latencies are not observable on the graph until the load on the server approaches its maximum capacity. At those points, latencies rise rapidly, but the slope of the response times changes once the server is in an overloaded state. Recall that response times are only tracked for successful calls.

Figures 7 and 8 shows the cumulative distributions of the response times measured at several loads for two sample configurations: stateful proxying using authentication with UDP and TCP, respectively. Note that the X axes are in log scale. An obvious and expected result is that, as the loads increase, the response times increase as well (i.e., the curves shift to the right on the graph). There are, however two other interesting features of the graphs.

First, curves tend to cluster in two clearly different regions of the Figures: One, towards the upper left of the graphs, and other, closer to the center and lower right. The characteristic that differentiates these two regions is whether the loads are below or above capacity, i.e., whether the system is under overload. We can see that, when overloaded, the response time distributions become significantly worse, and very quickly (i.e., not linearly in proportion to the load). For example, the stateful UDP auth configuration has a peak throughput of 700 calls/second, yet the gap between the 700 curve and the 800 curve is significant, especially considering the log scale. The TCP curve exhibits a similar gap between 400 and 500 calls/second, as the TCP configuration peaks at 400 CPS. This response time behavior is particularly important for SIP servers, which need to provide service quickly and smoothly, as they are used for real-time media such as voice and video.

Second, observe that several significant jumps occur in the UDP curve at certain response times (e.g., 64 ms, 500 ms, 1000 ms, etc.). The TCP curve, however, does not exhibit this behavior and is much smoother. These jumps are due to the various retransmission timers used by SIP for reliability when UDP is used as the transport protocol. SIP's primary packet retransmission timer, called Timer A, uses an exponential backoff starting at 500 milliseconds and doubles each subsequent time that it fires. When the system is overloaded, we see the manifestations of these timers firing by the jumps in response time at those timer values. This is because when SIP runs over TCP, it uses TCP's packet reliability and retransmission mechanisms rather than its own Timer A as is done with UDP. However, higher-level timers, e.g., the transaction timeout timer, are used with

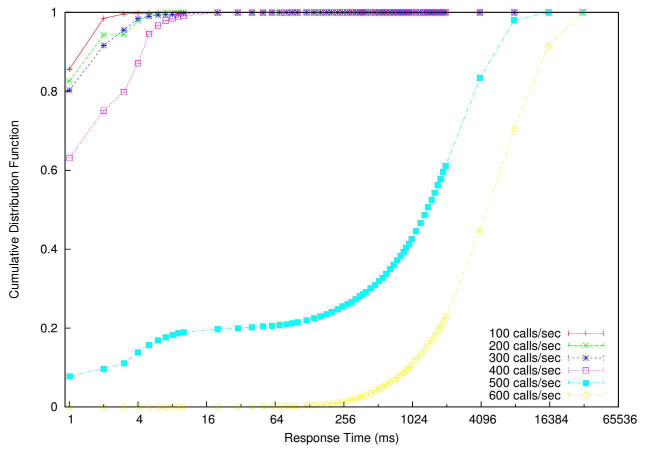

Figure 8: Response Time CDF: Stateful Proxying, TCP, with Authentication

both transport protocols.

5. RELATED WORK

Due to space limitations, we only briefly discuss related work in the SIP server performance area.

Janak's Thesis [4] describes many of the performance optimizations that are used by SER (and by implication, OpenSER). For example, rather than using zero terminated strings as defined by the C language, SER uses counted strings where the length of each string is stored with the string, making many operations constant time rather than linear based on the length of the string. SER also takes advantage of UTF-8 encoding to canonicalize certain headers for comparing in linear time, despite SIP's requirement to be case-insensitive. Finally, SER uses *lazy parsing* to only parse those headers necessary rather than naively parsing all headers, and *incremental parsing* to only scan needed fields within a header.

Salsano et al. [9] present an experimental performance analysis of SIP security mechanisms using an open source SIP proxy implemented in Java. They found that adding digest authentication to an `INVITE` transaction increases processing overhead by about 80% for a stateless proxy and 45% for a stateful proxy. They found minimal overhead using TCP or TLS instead of UDP. The server performance in these experiments was on the order of tens of calls per second. Given these relatively low numbers, we are not certain how representative these results are.

Cortes et al. [1] measured the performance of four transaction stateful SIP proxies using a suite of five tests. Proxy performance ranged from 90 to 700 calls per second. The tests used UDP only and evaluated parsing, string processing, memory allocation, thread overhead and overall capacity. Their results showed each of these components significantly affected performance, with parsing, string handling and memory management contributing from 33% to 88% of processing time.

6. SUMMARY

In this paper, we evaluate SIP proxy server performance, examining the impact of authentication and transport protocol on performance, as well as statelessness vs. statefulness. We study these issues experimentally, using OpenSER, a high-performance open-source SIP server, and SIPp, the de-facto standard for SIP performance benchmarking.

We find that performance varies widely, by an order of magnitude. Depending on the configuration (authentication enabled/disabled, UDP or TCP, stateful or stateless), throughput can vary from hundreds of operations a second to thousands. Authentication has the greatest impact across all configurations, due to the increased use of the database. TCP is more expensive than UDP for most configurations, and stateful proxying slower than stateful proxying. We show that latency distributions are highly influenced by the load, especially when the system is in an overloaded state.

Based on our results, we believe many potential future research issues exist, including examining other SIP server scenarios, optimizations identified by profiling, overload control, and using SSL/TLS as a transport.

The full scope of this work is available as an IBM Research Report [6]. A short 2-page extended abstract appears in [7].

Acknowledgments

Thanks to Arup Acharya and Henning Schulzrinne for many insightful discussions, and to Vijay A. Balasubrimaniyan for help with configuring OpenSER.

7. REFERENCES

[1] M. Cortes, J. R. Ensor, and J. O. Esteban. On SIP performance. *Bell Labs Technical Journal*, 9(3):155–172, Nov 2004.

[2] R. Gayraud and O. Jacques. SIPp. `http://sipp.sourceforge.net`.

[3] iptel.org. SIP express router (SER). `http://www.iptel.org/ser`.

[4] J. Janak. SIP server proxy effectiveness. Master's thesis, Czech Technical University Department of Computer Science, Prague, Czech Republic, May 2003.

[5] D. Malas. SIP performance metrics. Internet draft draft-malas-performance-metrics-06.txt (work in progress), IETF, Jan. 2007.

[6] E. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP server performance. Research Report 24183, IBM T.J. Watson Research Center, Feb. 2007.

[7] E. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP server performance (extended abstract). In *ACM SIGMETRICS*, San Diego, CA, June 2007.

[8] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[9] S. Salsano, L. Veltri, and D. Papalilo. SIP security issues: The SIP authentication procedure and its processing load. *IEEE Network*, pages 38–44, November/December 2002.

[10] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. RFC 3550, IETF, July 2003.

[11] The MySQL Project. The MySQL database server. `http://www.mysql.org`.

[12] www.openser.org. The open SIP express router (OpenSER). `http://www.openser.org`.