RC24183 (W0702-053) February 13, 2007
Computer Science

IBM Research Report

Evaluating SIP Server Performance

Erich Nahum, John Tracey, Charles P. Wright
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Evaluating SIP Server Performance

Erich Nahum, John Tracey, Charles P. Wright
IBM T. J. Watson Research Center
Hawthorne, NY, 10532

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 server performance using micro-benchmarks meant to capture common SIP server scenarios: registration, proxying, and redirection. We use standard open-source SIP software such as OpenSER and SIPp, running on an IBM BladeCenter with Red Hat Enterprise Linux with Gigabit Ethernet connectivity.

We show performance varies greatly depending on the server scenario and 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 can affect performance by a factor of 2-4. We also provide kernel and application profiles using Oprofile that help explain and illustrate processing costs. Finally, we provide a simple fix for transaction-stateful proxying that improves performance by a factor of 10.

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 servers perform under various scenarios and explain some of the limits to performance. We evaluate SIP server throughput and latency for several common SIP server scenarios, using micro-benchmarks on a dedicated experimental testbed.

In particular, we are interested in identifying the primary factors that determine SIP server performance including:

- How do different SIP server scenarios (registration, proxying, and redirection) compare in performance?
- 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 also provide profiling results which illustrate where the significant processing costs are incurred.

We find that SIP performance, in terms of throughput, can vary by an order of magnitude, depending on the scenario and how the server is configured. Registration, proxying, and redirection can each can vary from hundreds to thousands of operations per second, depending on whether authentication is used, whether transactions are 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. We also present a simple one-line performance fix to the transaction stateful timer code, identified by profiling, that improves stateful proxying by up to a factor of 10.

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 must need to be provisioned.

The rest of this paper is organized as follows: Section 2 provides more background on SIP and SIP servers. Section 3 presents the SIP server scenarios that we evaluate. Section 4 describes our experimental setup, and Section 5 presents our results in detail. Section 6 discusses some related work, and Section 7 provides our summary and conclusions and briefly presents plans for future work.

2 Background

This section provides a short general background on SIP required to understand and interpret our results. Those familiar with SIP may wish to skip to Section 3.

2.1 Protocol Overview

As described earlier, SIP is a control-plane protocol designed to create, modify, and destroy media sessions between two or more parties. RFC 3261 [20] is the core SIP IETF specification, although there are many RFCs that augment and extend the protocol. Sessions can be of various types, including voice, video, and text, which are carried over a separate data-plane protocol. This decoupling of the control plane from the data plane is one of the fundamental strengths of SIP and contributes to its flexibility and extensibility. SIP was designed with this in mind; for example, SIP application routers, known as proxies, are required to preserve and forward headers that they do not understand. Similarly, SIP runs over TCP, UDP, SCTP, SSL, IPv4 and IPv6. SIP is not a network resource reservation protocol such as RSVP [29]; the issue of how to allocate and manage network bandwidth and transfer media is outside the scope of the protocol. SIP is a text-based protocol that borrows much of its mechanics from HTTP [7]. Messages contain headers and optionally bodies, depending on the message type.

As an example, for voice applications, SIP messages encapsulate an additional protocol, the Session Description Protocol (SDP) [21], which provides an offer/answer model to negotiate session parameters, such as which voice codec to use. Once the end points agree to the session characteristics, voice data is typically carried by the Real-time Transport Protocol (RTP) [26]. Media protocols usually communicate directly between end points in a peer-to-peer fashion, although this can, of course, be complicated by factors such as firewalls and Network Address Translation (NAT). SIP can aid in traversing such network middleboxes, however.

2.2 SIP Users, User Agents, Servers

SIP users are usually identified using a SIP URI, e.g., `sip:user@domain.com`. This provides a layer of indirection that enables features such as mobility and location-independence.

SIP users employ end points typically referred to as *user agents.* These are the entities that initiate and accept sesions. They include both hardware (e.g., hard VoIP phones, cell phones, pagers, etc.) and software (e.g., soft phones, IM clients, voicemail servers). User agents are further categorized into *User Agent Clients* (UAC) and *User Agent Servers* (UAS), depending on how they are acting, either making requests (UAC) or responding to them (UAS). Thus most call flows for SIP messages include how the UAC and UAS behave for that scenario.

The SIP infrastructure provides several functional network elements in order to deliver messages to their ultimate users; these are the specialized servers that are the focus of this study. They include:

- *The Location Service* provides a service to find and update user location.
- *Registrars* provide the interface for users to register their location with the SIP infrastructure. Users can provide multiple locations. The registrar updates the location service with the appropriate information.
- *Proxies* route messages towards their eventual destinations. If a message is intended for a user that the proxy is responsible for (i.e., is in its domain), the location service is queried to determine the final destination for the message. If the proxy is not responsible for the user, it is fowarded to another proxy. The next proxy can be chosen via multiple methods, but a common way to decide is via DNS location [22].
- *Redirectors* send a redirect message back to the source of the message notifying it of the next place to send the message; the idea is for the redirect server to take itself ‘out of the loop’ in message forwarding. The concept is analogous to DNS servers providing redirect messages rather than recursive lookups.

The above services are described *functionally*. They can be co-located in practice. For example, the registrar and location service are frequently located together.

2.3 Sessions, Transactions and Dialogs

SIP consists of several layers, which define how the protocol is conceptually and functionally composed, but not necessarily implemented. The lowest layer is called the syntax/encoding layer, which defines how messages are constructed. This layer is conceptually above the IP transport layer, e.g., UDP or TCP. SIP encodings are specified using an augmented Bakcus-Naur Form grammar (ABNF). The second layer is called (perhaps unfortunately) the transport layer. This layer specifies how a SIP client sends requests and handle responses, and how a server receives requests and sends responses. The third layer is the transaction layer. This layer is responsible for matching each response to the corresponding request, SIP application-layer protocol timeouts, and retransmissions. All SIP end points (i.e. user agents) have transaction layers, as do stateful proxies. Stateless proxies do not. The fourth layer is called the transaction user (TU) layer, which may be thought of as the application layer in SIP. The TU creates an instance of a client request transaction and passes it to the transaction layer.

SIP uses HTTP-like request/response *transactions*. A transaction consists of a request to perform a particular method (e.g., `INVITE`, `BYE`, `CANCEL`, etc.) and at least one response to to that request. Responses may be *provisional*, namely, that they provide some short term feedback to the user (e.g., `TRYING`, `RINGING`) to indicate progress, or they can be final (e.g., `OK`, `401 UNAUTHORIZED`). Final responses complete the transaction; provisional responses do not.

A *transaction stateful proxy* is a proxy that maintains state for each transaction that passes through it, for both the client and the server. It will generate provisional responses (i.e., `TRYING`) to suppress retransmissions. A *stateless proxy* simply acts as a SIP message forwarder, and does not generate provisional responses.

A dialog is a SIP relationship between two user agents that endures for some period of time. Dialogs aid in message sequencing and routing between user agents, and provide a context in which to interpret messages. For example, in VoIP, an `INVITE` message not only creates a transaction (the sequence of messages for establishing the `INVITE`), but also, if the transactions completes successfully, a dialog (in this case, a phone call). A `BYE` message creates a new transaction and, when the transaction is complete, terminates the dialog. In this example, a phone call is the dialog, and the `INVITE` and `BYE` transactions deliniate it. Thus, SIP sessions consist of one or more dialogs, which in turn contain one or more transactions.

2.4 Authentication and Security

SIP security is complex, as described in the RFC [20], and is an area of ongoing research. For example, numerous RFCs and Internet-Drafts at the SIP working group page [12] focus on security issues. While the protocol does not address all potential security concerns, it does provide some mechanisms for confidentiality, authenticity and integrity.

SIP authentication is based on HTTP's MD5 Digest Authentication [8], providing not only verification of user identities but also protection against replay attacks. SIP servers may respond to requests with a challenge for the calling user agent (UAC) to authenticate itself. This is done via a `401 UNAUTHORIZED` (for registrars and redirectors) or `407 UNAUTHORIZED` (for proxies) response, which presents a challenge the UAC must respond to using the `Authorization:` header. The server challenge includes a nonce and the realm (e.g., domain) for the security region that the UAC wishes to authenticate itself for. The UAC response includes a credential which is derived from an MD5 hash of the nonce, realm, user name, and password. This allows the UAC to prove to the server that it knows the user's password without sending the password in the clear. While this mechanism allows authentication, it does not provide privacy; eavesdroppers can, for example, see who a UAC is calling.

Confidentiality and integrity can be provided by TLS [6], which is used in a hop-by-hop manner. Servers must hold a valid public key certificate that can be verified by user agents when the UAs establish connections to those servers. User agents may also hold certificates. Secure proxy-to-proxy communication requires that each side of the conversation verify the certificate of the other side. While TLS can have a substantial impact on server performance (see, for example, [3]), it is beyond the scope of this work. Similarly, SIP messages carry standard MIME bodies which may be encrypted using S/MIME [9], but we do not examine those issues here.

3 Scenarios

In this Section, we describe the three common SIP Server scenarios that we evaluate.

3.1 Registration

Registration is the process by which a SIP client (or user agent) notifies the SIP infrastructure where it can be located. This allows a SIP proxy to determine where messages for a particular SIP URI should be routed. We describe two cases of registration: with and without authentication.

Figure 1 shows a packet flow example for SIP registration with authentication enabled. The hashed circle

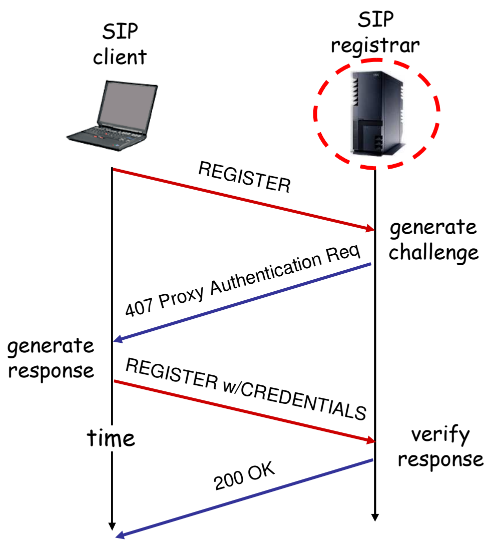

Figure 1: Registration with Authentication

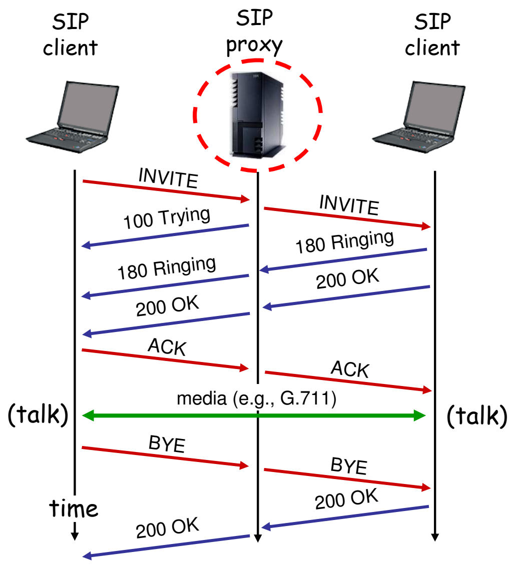

Figure 2: Stateful Proxying

surrounding the registrar indicates that this is the component we wish to measure in this scenario. In this example, the client sends a `REGISTER` message to the registrar containing its SIP URI, its contact IP address and port number, and an expiration time in seconds to specify the duration for which the registration information is valid. Because authentication is enabled, the registrar challenges the client to prove its identity using the `401 UNAUTHORIZED` response. The client then re-retransmits the `REGISTER` message with an additional `Authorization:` header that provides the newly formed credentials, as described in Section 2.4. The registrar then checks the response, and if correct, saves the contact information for that client's SIP URI and responds with a `200 OK` message.

In the scenario where registration is performed without authentication, the packet flow consists simply of the `REGISTER` message from the client followed by the `200 OK` from the Registrar. This case might occur in an enterprise environment where the network is secured and users are trusted.

3.2 Proxying

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 2 shows an example of stateful proxying without authentication. The hashed circle around the proxy again illustrates that this is the component 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 [26]. 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` response 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. However, SIP proxies can be configured so that not all messages need traverse the proxy. For ex-

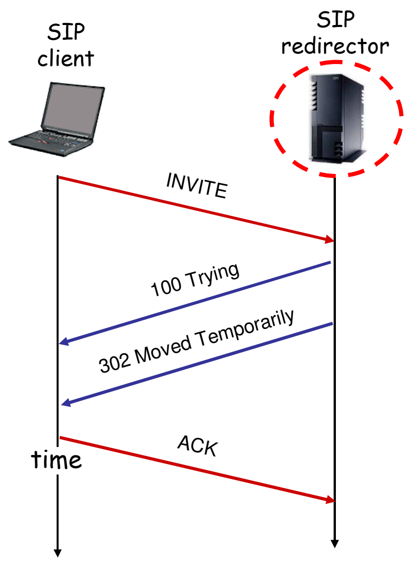

Figure 3: Redirection

ample, the BYE /200 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 2, 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, the proxy responds to the original INVITE with a 407 UNAUTHORIZED message, challenging the client to provide credentials that verify its claimed identity with a response based on the challenge. The client then retransmits the INVITE message with the generated credentials in the Authorization: header, analogous to the registration scenario shown in Figure 1. The proxy also challenges the 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.

3.3 Redirection

A redirect server is a user-agent server (UAS) that retrieves address information about reaching the callee but returns it to the caller, rather than forwarding a request as a proxy does. Calling clients are redirected to other servers using the 3XX response codes. A redirector is used to reduce processing load on proxy servers by redirecting the client to another server. This is analogous to a DNS server in redirect mode vs. recursive mode; the redirect server takes itself “out of the loop” of the forwarding path while still providing some information to the client. The client can then retransmit the request to the redirected address. Redirection is used for scalability; for example, to propagate URIs from the core of the network to the edges or to load balance across a cluster of servers.

Figure 3 shows an example of redirection. In this scenario, the redirector is operating without authentication (as a UAS, a redirect server is always transaction-stateful). The client sends the original INVITE to the redirector, which immediately responds with the provisional 100 TRYING response. The redirector then retrieves the location of the SIP URI from the location service, and sends that information to the client via a 302 MOVED TEMPORARILY message. Since the 302 message is not a provisional response, it must be acknowledged by the client using the ACK message. The client then sends the INVITE to the next destination provided by the 302 MOVED TEMPORARILY message, not shown in Figure 3 since it does not involve the redirector.

If the redirector is configured to use authentication, a 407 UNAUTHORIZED response is generated with a challenge, and the client must retransmit the INVITE with the proper credentials in the Authorization: header.

3.4 Choice of Transport Protocol

Since 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.

4 Experimental Testbed

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

4.1 SIP Server Software

We use the Open SIP Express Router (OpenSER) [28], a freely-available, open source SIP proxy server. OpenSER is a “fork” of the SIP Express Router (SER) [13], sharing much of its code base. Both proxies have 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 more active development, but we believe our results will hold with SER as well. We considered other proxies such as Columbia’s SIPd and the SIP Proxies available from sipfoundry.org, such as sipXProxy and reSIPprocate. However, these tools were either not freely available (SIPd is now licensed by SIPQuest) or did not seem to have the same user community behind them (sipXproxy and reSIProcate).

We used OpenSER version 1.1.0 for our experiments. We produced several configuration files depending on the scenarios that we wished to test, such as registration, proxying, redirection, and the options we wished to explore, such as with/without authentication, TCP/UDP, stateful/stateless, etc. In situations where a user database was required, we used MySQL [19] 4.1.12-3.RHEL4.1, which we populated with 10,000 unique user names and passwords. OpenSER was configured to use a write-back caching policy, to maintain client state across restarts but also to achieve close to in-memory DB performance.

4.2 SIP Client Workload Generator

We use the SIPp [10] SIP workload generator, another freely available open-source tool. SIPp is similar to SER in that it has an active user community and appears to be the most frequently used open-source SIP testing tool, in the same way that httperf [17] is the most commonly-used HTTP client workload generator. 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 several new flows that were not included with SIPp to handle registration and 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 did not use SIPStone [27], since it is not freely available and again the license is held by SIPQuest.

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 on SIPp 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. It is well-known that an open-loop workload generator is required to decouple the request rate from the reply rate [1, 17, 25], and thus guarantee the ability to generate overload. 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. Upon publication, we will make our SIPp improvements available to ensure reproducibility.

4.3 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. For application and kernel profiling we use the standard open-source Oprofile [18] tool, version 0.8.1-21.

4.4 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.

4.5 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 for each scenario, in terms of the number of the appropriate completed operations per second; e.g., registrations per second, calls per second, etc. Latency, however, is more complex, as it is defined based on the scenario. Latency is defined in the following ways:

- *Registration*. For registration, latency is the wall-clock time as seen by the client between the sending of the original `REGISTER` message and the eventual `200 OK` response. This is the Registration

Request Delay (RRD) as defined in the IETF SIPPING Working Group Performance Metrics Draft [16].

- *Proxying*. For proxying, latency is 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 [16], except that there the latency timer is stopped when a `180 RINGING` response is received. Since there is negligible delay between the two messages on the UAS, we believe the difference is minimal.
- *Redirection*. For redirection, latency is defined as the time between the sending of the `INVITE` and the receipt of the `302 MOVED TEMPORARILY` response. This is again similar to SRD in [16].

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. Oprofile is configured to report the default `GLOBAL_POWER_EVENT`, which reports time in which the processor is not stopped (i.e., non-idle profile events).

4.6 Restrictions, Limitations, and Scope

Note that our setup by no means covers the entire space of configurations for SIP. 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.

5 Results

In this Section we present our results in detail. Proxying is discussed in Section 5.1, registration in Section 5.2, and redirection in Section 5.3.

5.1 Proxying

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

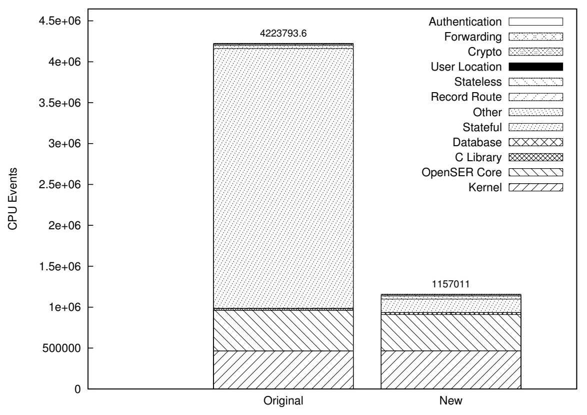

Figure 4: CPU Profiles: Transaction-Stateful Proxying, UDP, No Authentication, Load 1000 CPS

While examining the CPU profiling results generated using oprofile, one problem we observed very quickly was the extremely large amounts of CPU cycles being spent in the OpenSER module responsible for transaction-stateful processing. Figure 4 shows the CPU profile for the transaction-stateful proxying scenario without authentication running over UDP, as reported by oprofile. As can be seen, about 75 percent of the cycles are spent in the transaction-stateful module (the bar marked 'Original' in the Figure).

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 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. A one-line fix corrected this problem, reducing the cost from linear time based on the number of calls to constant time. Figure 4 also shows the CPU profile for the same configuration after the fix has been applied. Observe that the spent in the transaction stateful module falls dramatically.

Figure 5 shows how this fix improves performance. This graph shows offered versus achieved load for the transaction-stateful proxying scenario, both with and without authentication, for the original and the fixed version of OpenSER using UDP as the transport. Note that both the X and Y axes are in log scale. Throughputs increase with load until saturation is reached, then fall as the server is overloaded. Peak throughputs for each curve are also reported in Figure 6. Peak throughputs are calculated as the maximum throughput achieved while

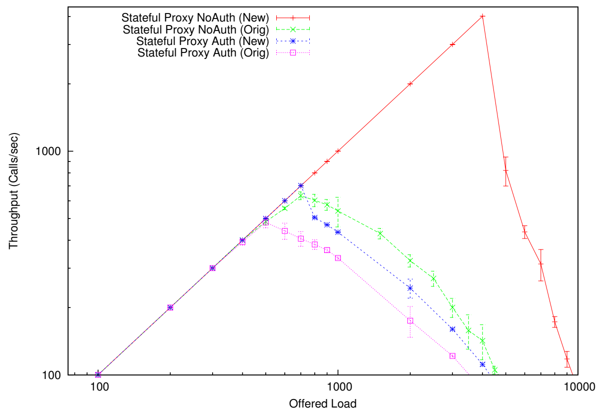

Figure 5: Throughputs vs. Offered Load: Transaction-Stateful Proxying, Original vs. New

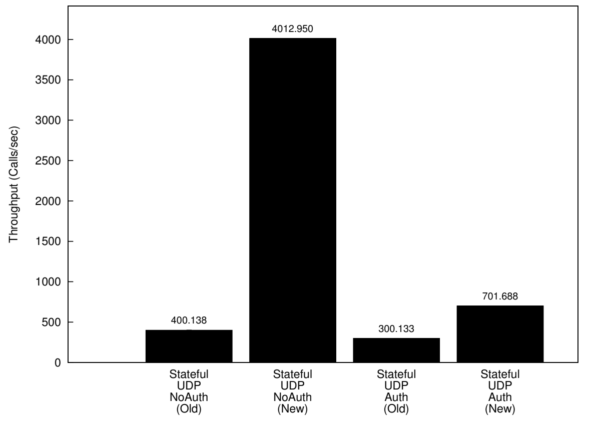

Figure 6: Peak Throughputs: Transaction-Stateful Proxying, Original vs. New

maintaining at least a 99 percent success rate. As can be seen, peak throughput increases by over a factor of ten in the scenario without authentication, from 400 to 4012 calls per second. Performance increases by 250 percent for the scenario with authentication, from 300 to 701.

In the case where authentication is used, the time spent in the transaction stateful module is less significant, only about 14 percent, yet a three-fold performance improvement is obtained. We believe that this is because of indirect effects due to cache behavior; by needlessly walking the timer chains, the original timer code evicted code and data used by other functions, causing them to take longer time than necessary.

Since transaction-stateful behavior is integral not only to proxying but also to registration and redirection, all subsequent results reported below include our timer fix.

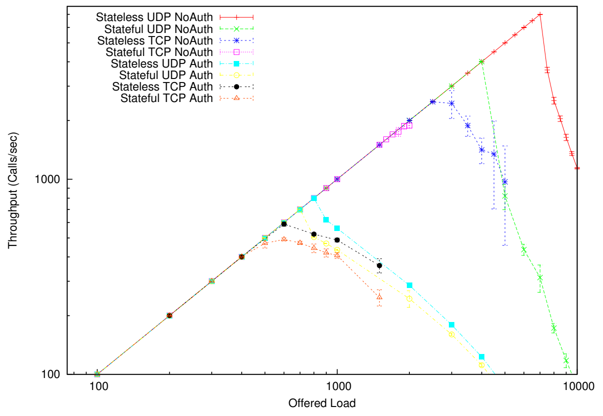

Figure 7: Throughputs vs. Offered Load: Proxying

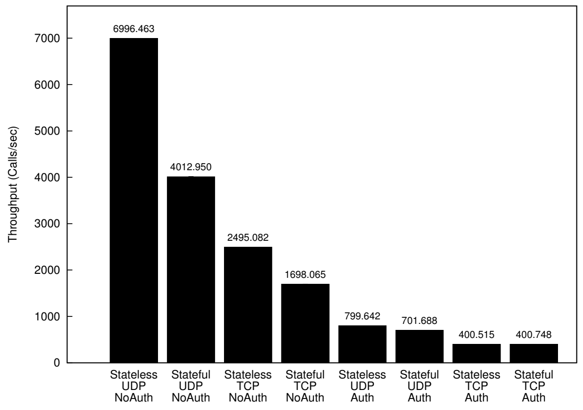

Figure 8: Peak Throughputs: Proxying

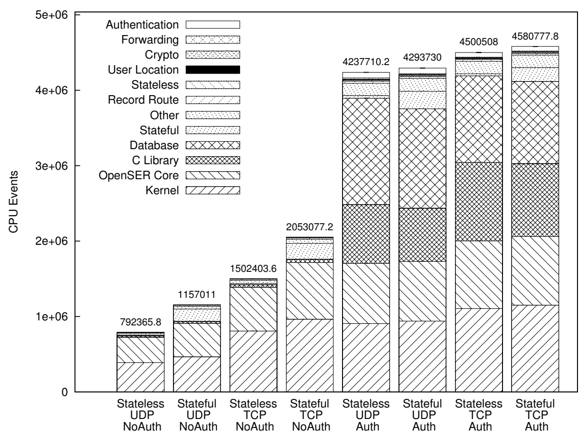

Figure 9: CPU Profiles: Proxying

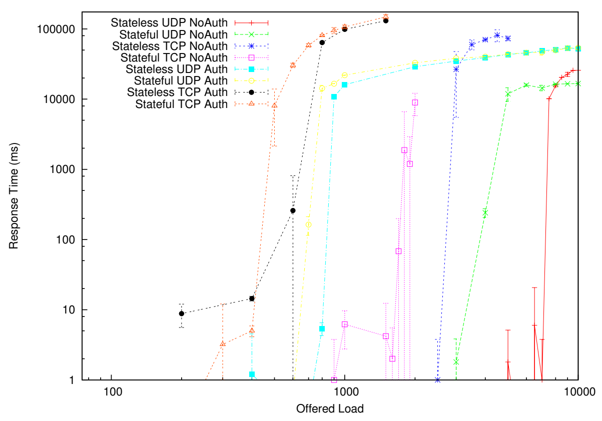

Figure 10: Average Response Time: Proxying

Throughputs

Figure 7 shows throughputs versus offered load for stateful and stateless proxying, with and without authentication, using both UDP and TCP as transport protocols. Again, X and Y axes are in log scale. Peak throughputs for each curve are also reported in Figure 8. Peak throughputs are calculated as the maximum throughput achieved while maintaining at least a 99 percent success rate. Figure 9 presents the CPU profiles for the 8 configurations at a load of 1000 CPS.

As can be seen in Figures 7 and 8, 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). Figure 9 illustrates why performance degrades with authentication. Observe that when authentication is enabled, 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 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 4.1. 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 9, 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 10 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

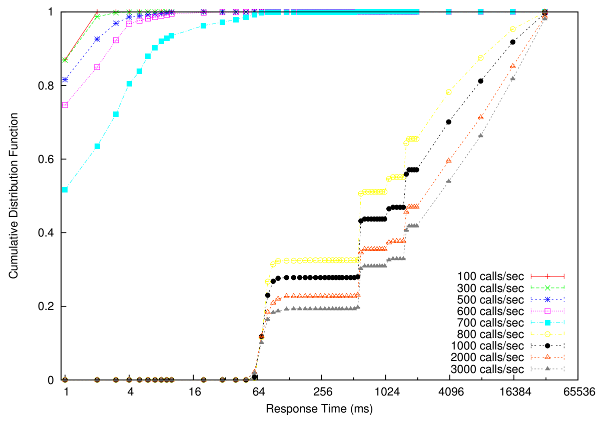

Figure 11: Response Time CDF: Stateful Proxying, UDP, with Authentication

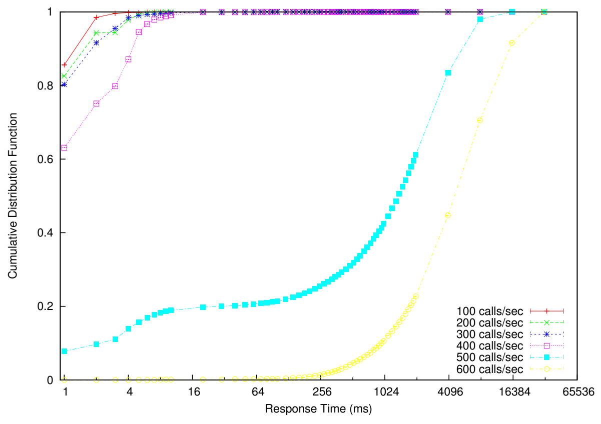

Figure 12: Response Time CDF: Stateful Proxying, TCP, with Authentication

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 11 and 12 show 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 axis is 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, 2000 ms, etc.). The TCP curve, however, does not show these jumps, and is much smoother. These 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,

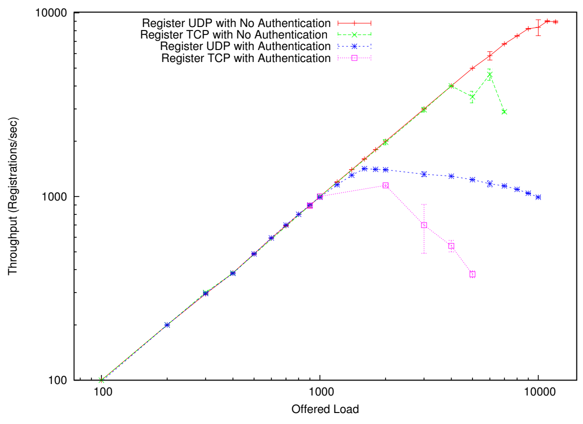

Figure 13: Throughputs: Registration

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. We do not see this behavior in the TCP graph, however. This is because when SIP runs over TCP, it leverages TCP's packet reliability and retransmission behavior rather than using its own as is done with UDP. However, higher-level timers, e.g., the transaction timeout timer, are used with both transport protocols.

Both of these above phenomena are consistent across all our configurations, not only for Figures 11 and 12. Due to space constraints, not all our data can be shown. However, more data is available for the interested reader in the Appendix.

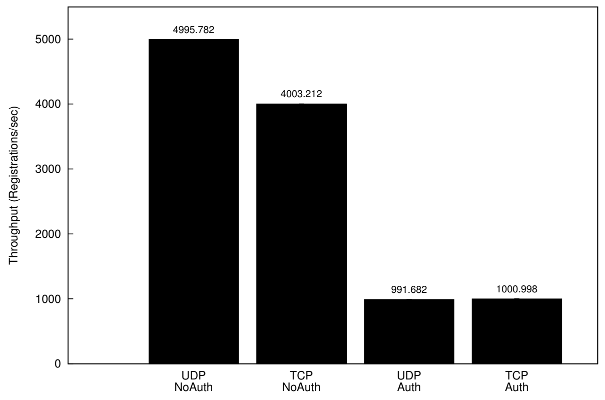

Figure 14: Peak Throughputs: Registration

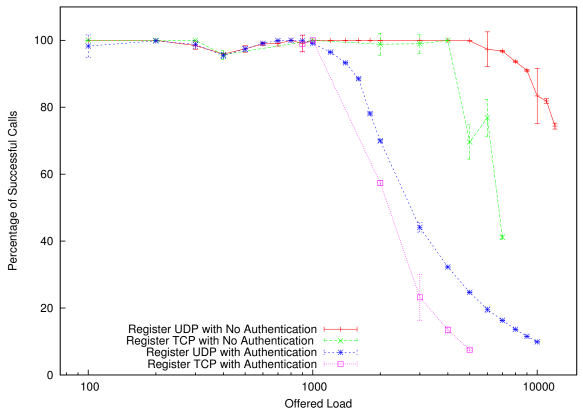

Figure 15: Success Rate: Registration

5.2 Registration

Throughputs

Figure 13 shows throughput versus offered load for registration, with and without authentication, with both UDP and TCP. As in Section 5.1, both X and Y axes are in log scale. Peak throughput is shown in Figure 14, again defined as the highest maximum throughput achieved while maintaining a 99% registraton success rate. Figure 15 shows the percentage of successful registrations versus offered load. Figure 16 presents the CPU profiles for all four registration configurations, given a load of 1000 registrations/second.

We see that again, authentication has the most significant impact on registration performance, as with the proxying results in Section 5.1. Using authentication decreases peak throughput by 80% for UDP and 75% for TCP. The CPU profiles in Figure 16 again illustrate why: CPU usage increases by a factor of 2.5 when authentication is enabled. Of this increase, 58.4% can be attributed to the increase in the database and C library. The other major contributors to the increase are the OpenSER core and kernel, which account for 11.7% and 17.7% respectively. Again, the MD5 authentication only contributes 2.0% of the increase (less than 1% of the total of the authentication case).

If we use UDP without authentication as a baseline to compare the CPU profiles for the registration case with the proxying case, we can see notable differences. The OpenSER core module contributes 38% of the cycles for proxying, but contributes just 7.0% to the registration case. Similarly, the kernel contributes 40.3% of the cycles for proxying, but just 21.1% for registration. Instead, the database contributes 29.6% of the CPU cycles even without authentication. This is because each successful registration must update the database, as a `REGISTER` message updates the registration timeout value. As observed in the proxy case, the database

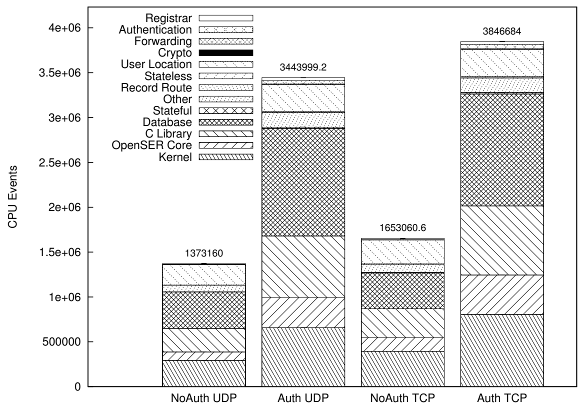

Figure 16: CPU Profiles: Registration

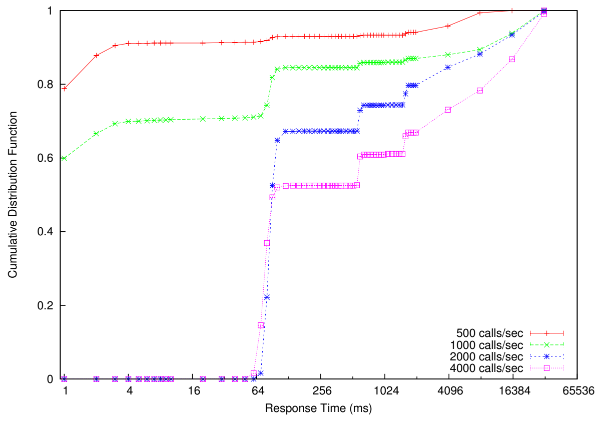

Figure 17: Response Time CDF: Registration with Authentication

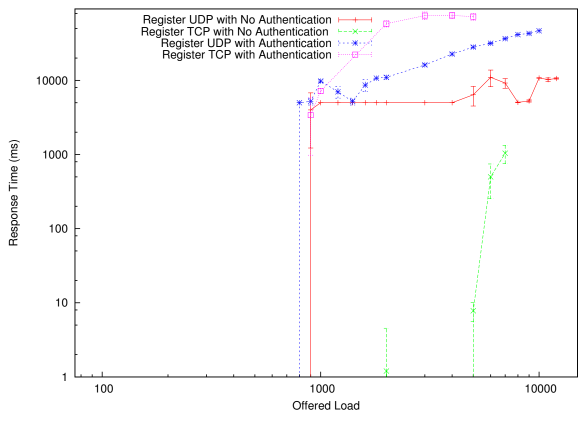

Figure 18: Average Response Time: Registration

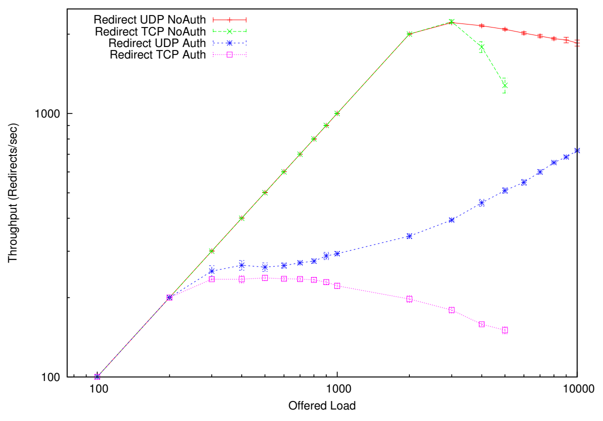

Figure 19: Throughput: Redirection

also makes extensive use of the C library, contributing 19.1% of the CPU cycles versus only 2.1% for proxying without authentication. Finally, the user location module, which is not used for proxying, contributes 16.5% to registration.

The choice of transport protocol, however, has less effect on the registration results than it does for the proxying results, with a drop of only 19.8% when authentication is not used, and virtually no effect when authentication is enabled. The transport protocol affects the registration scenario less than for proxying, because proxying is fundamentally more network oriented: minimal processing is done to the packet to determine the destination of the message, which is immediately forwarded to another host. In registration, however, the message must be completely parsed and the database updated.

Latencies

Registration average response times (shown in Figure 18) exhibit the same behavior as those from the proxying scenario in Section 5.1. Again, latencies are low until the server load approaches maximum capacity. At those points, latencies rise rapidly, but the slope of the response times changes once the server is in an overloaded state.

Figure 17 shows the cumulative distributions of the response times measured for several loads, using authentication and UDP. As with the proxying CDFs, there are several significant jumps occurring at certain response times due to retransmission timers. Unlike the proxy curves, these jumps occur even before overload (e.g., they are visible in the 800 CPS curve); and after overload the response time does not increase as quickly (e.g., under 1200 registrations per second, 52.8% of the response times are less than 1 ms).

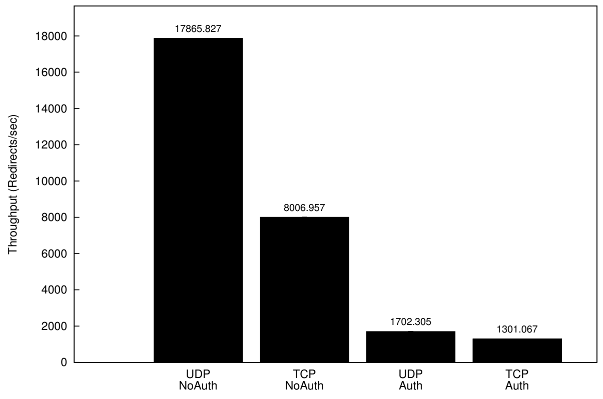

Figure 20: Max Throughputs: Redirection

5.3 Redirection

Throughputs

Figure 19 shows throughputs versus offered load for redirection with and without authentication, using both UDP and TCP. As in previous sections, both the X and Y axes are in log scale. Peak throughput is shown in Figure 20, again calculated as the highest maximum throughput achieved while maintaining a 99% success rate. Figure 21 presents the CPU profiles for all four configurations given a load of 1000 redirects/second.

The redirection scenario is able to achieve higher throughput than both the proxying and registration scenarios under all four of its configurations. The throughput is higher than that of proxying because fewer messages are exchanged. Even though redirection exchanges the same number of messages as registration, it is less computationally intensive and thus achieves a higher throughput. This result is expected as redirectors are designed to push routing logic to the user-agents in order to reduce the load on network elements.

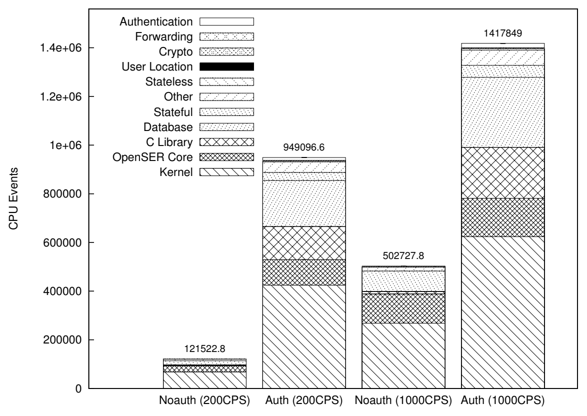

Figure 21: CPU Profiles: Redirection

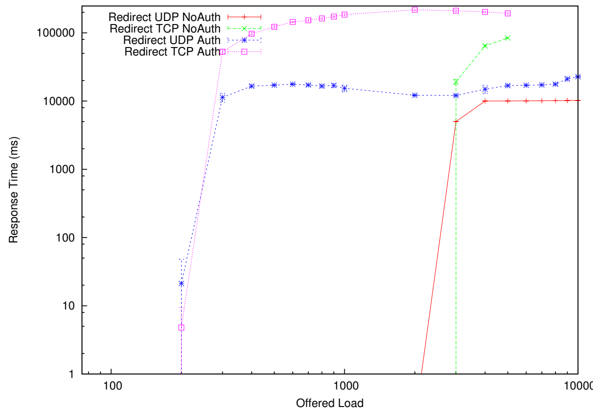

Figure 22: Average Response Time: Redirection

The factors that affect proxying and registration also influence redirection. Authentication has the largest impact on throughput, degrading performance by 90.5% for UDP and 83.7% for TCP. The choice of transport protocol is also significant: TCP is 23.5% and 55.2% slower than UDP with and without authentication, respectively. The reasons are the same as in the earlier scenarios: authentication requires significant processing resources for database accesses; and TCP results in increased kernel and OpenSER core processing.

Latencies

Redirection latencies, shown in Figure 22, exhibit the same behaviors as those in Sections 5.1 and 5.2. Latencies are not observable due to SIPp's 1 millisecond at loads under max capacity, and again rise steeply as capacity is reached. In general, response times using authentication are higher than without, and higher using TCP than with UDP. These results are expected given the capacities (service rates) of the respective configurations.

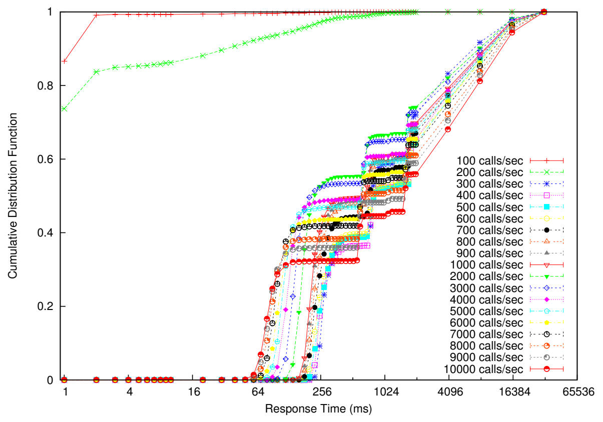

Figure 23: Response Time CDF: Redirection with UDP and Authentication

6 Related Work

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

Janak's Thesis [14] describes many of the performance optimizations that are used by SER (and by implication, OpenSER). For example, rather than using null 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 those fields within a header that are needed.

SIPStone [27] is an early SIP benchmark designed to evaluate SIP registrar, redirect and proxy servers. The benchmark consists of five interactions: registration, redirect, `INVITE` through an outbound proxy, and `INVITE` 's to an end proxy with both `408 TEMPORARILY UNAVAILABLE` and `200 OK` responses. Each interaction is performed using both UDP and TCP (using persistent connections) with an open loop workload generator. The SIPStone tech report defines and implements a benchmark but does not evaluate any servers.

Salsano et al. [24] include an experimental performance analysis of SIP security mechanisms using an open source Java SIP proxy server. For the specific SIP proxy and hardware tested (300 MHz Pentium, 128 MB RAM, 100 Mb Ethernet), adding digest authentication to an `INVITE` transaction increases processing overhead by about 80% for a stateless proxy and 45% for a stateful proxy. Ignoring overhead associated with connection setup, use of TCP incurs minimal overhead relative to UDP, and addition of TLS incurs negligible overhead. The total capacity of the server in these experiments was on the order of tens of INVITES per second. Given these relatively low numbers, we are not sure how representative these results are.

Cortes et al. [5] measured the capacity of four transaction stateful SIP proxies using a suite of five tests. The tests used UDP only and evaluated parsing, string processing, memory allocation, thread overhead and overall capacity. Their results showed each of the four individual components significantly affected capacity, with parsing, string handling and memory management contributing from 33% to 88% of processing time. Parsing was shown to consume roughly a quarter of processing time despite large variations in absolute values. Other elements were shown to vary widely among proxies, with string processing ranging from 7% to 45% and memory allocation varying from 3% to 14%. For the 450 MHz dual-SPARC processors on which the tests were performed, maximum capacity of the proxies ranged from 90 to 700 calls per second.

7 Summary

In this paper, we evaluate SIP server performance for three core SIP server scenarios: proxying, registration, and redirection. We also examine the impact of authentication and transport protocol on performance, as well as statelessness vs. statefulness for proxy servers. We study these issues experimentally, using OpenSER, a high-performance open-source SIP server, and SIPp, the de-facto standard for SIP performance benchmarking. For our experiments, we use an Intel Xeon processor-based blade as our server, running the Linux operating system, connected over a private network to several Intel-based client workload generators.

We find that performance varies widely, by an order of magnitude. Depending on the scenario (proxying, redirection, registration) and on the configuration (authentication enabled/disabled, UDP or TCP, stateful or stateless), throughput can vary from hundreds of operations a second to tens of thousands. Authentication has the greatest impact across all scenarios, 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. Finally, we provide a simple fix for stateful proxying which improves performance by up to a factor of 10.

Based on our results, we believe many potential future research issues exist, including:

- *OpenSER and Linux optimizations*. Profiling has revealed many opportunities to improve performance for SIP servers, in both the server and the kernel.
- *Overload control.* SIP servers need to be able to maintain throughput under overload. This also an active area of research in IETF (e.g., [11]). Investigating various approaches to admission control, load shedding or service degradation seems appropriate.
- *User-model based benchmarking.* This work has taken a micro-benchmark approach to evaluating performance. Previous work in Web servers (e.g., SURGE [2] and SPECWeb2005 [4]) has shown that how users interact with a system can have a significant impact on performance. A straightforward next step would be to incorporate user activity into the workload generation, e.g., talk durations and ringing times.
- *DB Performance.* Given the impact of the database on SIP server performance, several avenues could be pursued: varying the DB caching policy, using a customized DB rather than a general-purpose one, or evaluating locating the DB on a separate machine.
- *Instant messaging and presence.* Our work has focused on VoIP-based scenarios. Given the rise in the use of SIP for instant messaging and presence, how those systems perform will be necessary for provisioning and dimensioning those services.
- *SSL/TLS as a transport.* Given the heavy costs of SSL/TLS on Web server performance [3], it seems likely that the same will be true for SIP servers. Evaluating and understanding this cost will be necessary if SIP servers are to provide confidentiality.

We plan to explore many of the areas described above.

Acknowledgments

Thanks to Arup Acharya and Henning Schulzrinne for many insightful discussions about SIP and SIP server performance, and to Vijay A. Balasubrimaniyan for help with OpenSER configuration.

References

[1] Gaurav Banga and Peter Druschel. Measuring the capacity of a Web server. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[2] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[3] Cristian Coarfa, Peter Druschel, and Dan S. Wallach. Performance analysis of TLS Web servers. *ACM Transactions on Computer Systems*, 24(1):39–69, 2006.

[4] The Standard Performance Evaluation Corporation. SPECWeb2005. `http://www.spec.org/osg/web2005`, 2005.

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

[6] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[8] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, Internet Engineering Task Force, June 1999.

[9] J. Galvin, S. Murphy, S. Crocker, and N. Freed. Security multiparts for MIME: multipart/signed and multipart/encrypted. RFC 1847, Internet Engineering Task Force, October 1995.

[10] Richard Gayraud and Olivier Jacques. SIPp. `http://sipp.sourceforge.net`.

[11] V. Hilt, D. Malas, I. Widjaja, and R. Terpstra. Session initiation protocol (SIP) overload control. Internet draft draft-hilt-sipping-overload-00.txt (work in progress), Internet Engineering Task Force, October 2006.

[12] Internet Engineering Task Force (IETF). SIP working group charter. `http://www.ietf.org/html.charters/sip-charter.html`.

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

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

[15] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session initiation protocol (SIP) basic call flow examples. RFC 3665, Internet Engineering Task Force, December 2003.

[16] Daryl Malas. SIP performance metrics. Internet draft draft-malas-performance-metrics-05.txt (work in progress), Internet Engineering Task Force, September 2006.

[17] David Mosberger and Tai Jin. httperf – a tool for measuring Web server performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.

[18] OProfile. A system profiler for Linux. http://oprofile.sourceforge.net/.

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

[20] 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.

[21] J. Rosenberg and Henning Schulzrinne. An offer/answer model with session description protocol (SDP). RFC 3264, Internet Engineering Task Force, June 2002.

[22] J. Rosenberg and Henning Schulzrinne. Session initiation protocol (SIP): locating SIP servers. RFC 3263, Internet Engineering Task Force, June 2002.

[23] Jonathan Rosenberg, Henning Schulzrinne, and Gonzalo Camarillo. The stream control transmission protocol (SCTP) as a transport for the session initiation protocol (SIP). RFC 4168, Internet Engineering Task Force, October 2005.

[24] Stefano Salsano, Luca Veltri, and Donald Papalilo. SIP security issues: The SIP authentication procedure and its processing load. *IEEE Network*, pages 38–44, November/December 2002.

[25] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Closed versus open loop system models: Understanding their impact on performance evaluation and system design. In *Network Systems Design and Implementation (NSDI)*, San Diego, California, May 2006.

[26] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. RTP: a transport protocol for real-time applications. RFC 3550, Internet Engineering Task Force, July 2003.

[27] Henning Schulzrinne, Sankaran Narayanan, Jonathan Lennox, and Michael Doyle. SIPstone - benchmarking SIP server performance. `http://www.sipstone.org`, April 2002.

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

[29] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and David Zappola. RSVP: a new resource reservation protocol. *IEEE Communications Magazine*, 40(5):116–127, May 2002.