Improving Authentication Performance of Distributed SIP Proxies

Italo Dacosta, Vijay Balasubramaniyan, Mustaque Ahamad and Patrick Traynor
Converging Infrastructure Security (CISEC) Laboratory
Georgia Tech Information Security Center (GTISC)
School of Computer Science, Georgia Institute of Technology
Atlanta, GA, 30332
{idacosta, vijayab, mustaq, traynor}@cc.gatech.edu

ABSTRACT

The performance of SIP proxies is critical for the robust operation of many applications. However, the use of even light-weight authentication schemes can significantly degrade throughput in these systems. In particular, systems in which multiple proxies share a remote authentication database can experience reduced performance due to latency. In this paper, we investigate how the application of parallel execution and batching can be used to maximize throughput while carefully balancing demands for bandwidth and call failure rates. Through the use of a modified version of OpenSER, a high-performance SIP proxy, we demonstrate that the traditional recommendation of simply launching a large number of parallel processes not only incurs substantial overhead and increases dropped calls, but can actually decrease call throughput. An alternative technique that we implement, request batching, fails to achieve similarly high proxy throughput. Through a carefully selected mix of batching and parallelization, we reduce the bandwidth required to maximize authenticated signaling throughput by the proxy by more than 75%. This mix also keeps the call loss rates below 1% at peak performance. Through this, we significantly reduce the cost and increase the throughput of authentication for large-scale networks supporting SIP applications.

Categories and Subject Descriptors

C.2.0 [**Computers-Communication Networks**]: General—*Security and protection*

General Terms

SIP, authentication

Keywords

telecommunications, proxy, digest authentication, performance

1. INTRODUCTION

The *Session Initiation Protocol* (SIP) is an application layer signaling framework. Instead of simply connecting participants, SIP allows users to negotiate the terms of, establish and terminate sessions. Because of its relative simplicity (i.e., human readable encoding) and flexibility (i.e., transport layer agnosticism), SIP is now used by a wide array of multimedia services including video conferencing, instant messaging and presence.

SIP is also being widely adopted by IP telephony providers, including Vonage and AT&T, as the foundation of large scale deployments. Such providers typically deploy SIP proxies, nodes responsible for routing call requests and assisting in billing operations, across multiple geographic regions in order to efficiently deal with substantial volumes of traffic. With the assistance of a centralized server, these proxies also aid in the authentication of users and signaling requests. This architecture is often used to protect backend databases from attack and to minimize distributed consistency issues. In the face of increasing volumes of both legitimate and malicious (i.e., spam) traffic at currently deployed nodes, developing effective techniques for scaling proxy throughput is becoming a critical task.

In this paper, we present an empirical study of the impact of two techniques designed to improve the throughput of authenticated signaling requests for distributed SIP proxies. Our results show that an enhanced version of the parallel process execution currently recommended can dramatically increase proxy throughput. This improvement comes at the cost of a significant bandwidth overhead (≈ 25 Mbps) and an increased dropped call rate. Request batching, an alternative technique that has not previously been applied in this domain, is then shown to greatly reduce the bandwidth costs but fails to achieve the same high throughput as the first technique. We demonstrate that a carefully balanced hybrid of these two approaches can achieve throughput at the proxy equal to the capacity of the authentication service, an improvement of 33% over the best parallel execution approach, using 77.3% and 76.6% less bandwidth for requests and responses between proxy and database and maintaining call dropping rates below 1%. We note that these improvements are not simply the result of maximizing the settings of the two mechanisms and that the careless combination of these techniques can actually degrade performance.

In so doing, we make the following contributions:

- **Modify and extend OpenSER to support batch requests:** We made several enhancements to the open source OpenSER proxy software to allow for far greater throughput. We discuss our modifications and make them available as a patch to the community.

- **Characterize performance improvements of both parallelization and batching:** We performed an extensive measurement study of the performance gains realized by the use of multiple proxy processes, the use of batching and the range of combinations of these two approaches. Our analysis offers the first discussion of the tradeoffs inherent to these performance enhancing techniques in the context of SIP proxies.

- **Provide recommendations for optimal throughput and resource use by proxies:** By performing our experiments over a range of possible configurations, we are able to demonstrate that throughput is not maximized simply by using the maximum possible settings for parallelization and batching. Instead, carefully tuning settings can maximize throughput while using approximately 24% of the bandwidth required in naïve settings.

The remainder of this paper is organized as follows: Section 2 discusses the results of a number of related studies; Section 3 gives an overview of SIP and digest authentication; Section 4 describes our methodology and provides the details of our experimental testbed; Section 5 provides the results of our experiments for multiple possible configurations; Section 6 discusses additional considerations and identifies a number of future contributions that can be made; Section 7 offers concluding remarks.

2. RELATED WORK

Performance and scalability are critical properties of telecommunication networks. Emergency situations (i.e., natural disasters, terrorist attacks [12]), large scale events (i.e., American Idol, President's Inauguration Day [20]), and Denial of Service attacks [15] are examples of events that can cripple the availability of the telecommunication network. SIP, as one of the central protocols of many IP telephony networks, has an important role in the performance and scalability of such systems.

The performance and scalability of a SIP infrastructure have been the subject of several studies. Schulzrinne et al. [23] describe a set of metrics for evaluating and benchmarking the performance of SIP proxy, redirect and registrar servers, and an architecture and techniques to measure SIP server performance. Other studies have focused exclusively on the performance of a SIP proxy, analyzing how different configurations impact overall performance. Nahum et al. [17] evaluate the impact of state management, transport protocol and authentication on proxy's performance. Their findings show that proxy's performance varies greatly depending of the configuration chosen, and that authentication has the greatest impact across all the configurations analyzed. Salsano et al. [22] present a similar study focused on security configurations such as digest authentication and TLS. They also conclude that authentication considerably affects the performance of the proxy. Ono et al. [18] analyze how the use of TCP affects the performance and scalability of a proxy proposing several configuration changes to improve performance. Cortes et al. [8] evaluate the parsing, string processing, memory allocation, thread overhead and overall capacity of a proxy. While some of the previous studies have analyzed the impact of digest authentication on the proxy's performance, none of them consider scenarios where the user authentication credentials are stored in a remote database or a network authentication service is used instead (i.e., RADIUS). These scenarios are common in real deployments because they allow better scalability and therefore, are studied in this paper.

Several studies have suggested modifications and mechanisms to improve the performance of a SIP proxy. Janak [13] proposes the use of lazy parsing (parse only the minimum number of headers required) and incremental parsing (parse contents of headers incrementally) as significant performance optimizations for a SIP proxy. Balasubramaniyan et al. [5] propose a dynamic distribution algorithm to continue to maintain state and yet improve throughput in a proxy farm. Cortes et al. [9] propose that SIP proxies in an IMS infrastructure should be configured with stateless and stateful functionality, allowing them to adapt to network and CPU conditions dynamically. Singh et al [25, 26, 27] present a reliable, scalable and interoperable Internet telephony architecture for user registration, call routing, conferencing and unified messaging using commodity hardware. In addition, they apply some of the existing web server redundancy techniques for high service availability and scalability to the VoIP SIP context, proposing an identifier-based two-stage load sharing method. Shen et al. [24] propose three new window-based feedback algorithms for overload controls in a proxy, and compare them with existent algorithms. However, none of these works have proposed a more efficient alternative to the use of digest authentication, even though it has been shown that it has a considerable impact on proxy's performance, specially if a remote database is used.

In this paper we evaluate the use of multiple processes and batch requests to improve the performance of a SIP proxy configured with digest authentication and a remote database. The use of batch queries has been explored in other areas. In file systems such as XFS [28] and FARSITE [4] batching file updates has been shown to improve performance significantly. Batch queries have also been used in business applications [16]. Boneh and Shacham [7] present how batching the SSL handshake protocol can improve SSL's performance on a web server. Our work is the first to apply the concept of batch requests in a SIP proxy to improve performance. We also show the benefits and disadvantages of using batch requests when compared with a multiple processes approach.

3. SIP BACKGROUND

In this section, we provide a brief overview of SIP and digest authentication.

3.1 SIP Components and Architecture

SIP is an application-layer signaling protocol. Instead of simply passing content between parties, SIP allows endpoints to negotiate the characteristics and protocols of communication and establish and tear down sessions [21]. While widely used to perform the signaling operations for multimedia applications including Voice over IP (VoIP) and video conferencing, SIP can also be used to establish connections for applications such as instant messaging. Because of the predominance of telephony related applications in SIP, we will select our examples from this domain.

A SIP network contains a number of different components. Instead of traditional telephones, SIP users communicate through *User Agents* (UAs). UAs are software entities running on a user platform (e.g., desktops, laptops, SIP phones). UAs are simply communications endpoints and are responsible for initiating, responding to and terminating calls between other endpoints. Because users and their UAs can move between different domains, SIP networks operate *SIP Registrars*. Like Home Agents (HAs) in Mobile IP or Home Location Registers (HLRs) in cellular telephony, SIP Registrars keep track of the current location (IP address) of a UA and are queried when attempting to establish a call. When a UA attempts to make a call, it relies on a *SIP Proxy* to route its request to the appropriate domain. SIP Proxies also assist in authentication and billing operations. Accordingly, the performance of these devices is of critical importance to providers. Because both proxy

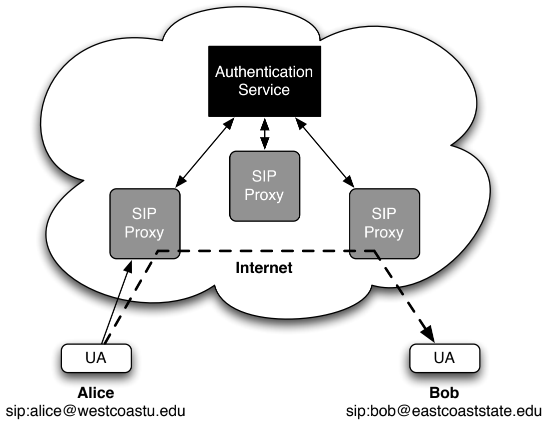

Figure 1: Basic SIP Infrastructure. Alice and Bob communicate via a VoIP provider with proxies distributed across the United States. These proxies share a single authentication server/cluster, located somewhere in the middle of the country. A similar architecture is becoming increasingly popular in cellular networks.

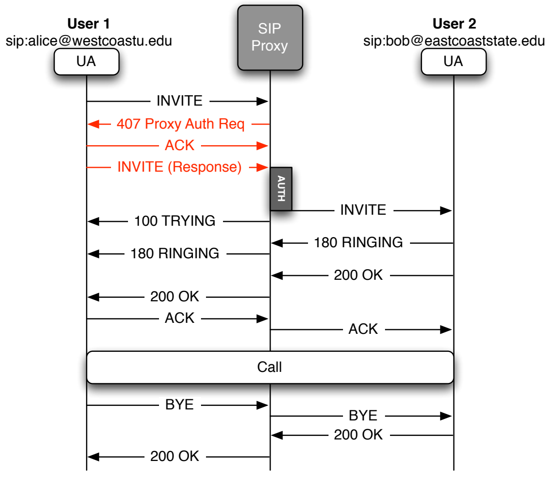

Figure 2: SIP call setup flow using digest authentication (in red). We remove the second proxy shown in Figure 1 to simplify our explanation.

and registrar functionality are implemented in software, these tasks are often executed on a single physical node. We therefore refer to such nodes as proxies for the remainder of this paper.

Figure 1 provides a high-level view of hypothetical nation-wide SIP network. A user Alice (alice@westcoastu.edu) in California attempts to call her friend Bob (bob@eascoaststate.edu) in Georgia. Notice that the provider deployed multiple proxies across the country to minimize the latency associated with servicing requests. However, as previously mentioned, the provider relies upon a centralized service to authenticate incoming call requests. Alice's call request (INVITE) is first transmitted to the closest SIP Proxy. The proxy then authenticates Alice's identity with the assistance of the centralized service. Successfully authenticated call requests are forwarded on to the SIP Proxy currently serving Bob. After the call is established, Alice and Bob directly exchange audio packets.

3.2 Digest Authentication

There are a number of ways in which users can be forced to authenticate themselves to a SIP network. Users could be required to execute protocols such as Kerberos or those using public keys (e.g., SSL). While these options offer strong security guarantees, they also carry significant overhead. A less strong but lighter-weight protocol based on HTTP-digest authentication [11], SIP digest authentication, is implemented by most available SIP UAs. Because this is the dominant scheme used by SIP clients, we investigate its characteristics further.

Figure 2 demonstrates a standard SIP call establishment with digest authentication interposed. When Alice attempts to call Bob, her UA transmits an INVITE message to her SIP Proxy. The proxy notes that the message must be authenticated to continue, generates a nonce n and returns the request to Alice's UA as a 407 (Proxy Authentication Request) message. Alice's UA acknowledges receiving the authentication request, computes the correct response and transmits it to the proxy. The proxy then requests the same answer from the authentication service, compares the two values and on match, forwards the INVITE message.

The SIP Digest Authentication response is computed as follows:

$$DA = MD5(HA1|n|nc|cnonce|qop|HA2)$$

where the MD5 hash algorithm is run over the concatenation of a value HA1, the nonce generated by the proxy, an optional nonce count (nc) value to prevent replay attacks, an optional client nonce ($cnonce$), the Quality of Protection (qop) that should be set to "auth" and a value HA2. HA1 and HA2 are calculated as follows:

$$\begin{aligned} HA1 &= MD5(username|realm|password) \\ HA2 &= MD5(method|digestURI) \end{aligned}$$

where $username$ and $password$ are unique values to Alice (and ideally all other users), $realm$ is the protection domain (i.e., the provider or proxy's name), $method$ is the SIP action being authenticated and $digestURI$ is the destination address the client is attempting to reach.

Digest authentication has a number of weaknesses. Most significantly, it is vulnerable to man in the middle attacks as it lacks a means of mutual authentication. Moreover, a number of weaknesses in MD5 [29, 6, 14] have lead the security community to recommend the use of other hashing algorithms. Finally, this technique may also be susceptible to efficient offline attacks such as password guessing. In spite of these observations, MD5-based Digest Authentication is still the most widely deployed authentication mechanism for SIP UAs and is still considered safe as attacks against high-entropy passwords have not yet been successfully executed.

4. EXPERIMENTAL SETUP

To characterize the performance of a SIP proxy with digest authentication, we implemented an experimental test bed based on the scenario depicted in Figure 1. SIP proxies are located in areas geographically close to subscribers to reduce effects of network latency on session setup. The proxies share a central authentication service which stores subscriber information and performs authentication related operations.

The authentication service can perform the authentication operations itself (i.e., RADIUS, LDAP) or can simply store subscriber information. In the latter case, the authentication service sends the subscriber's authentication credentials to the proxies, and the proxies perform the authentication operations and then delete the credentials. This is the model used in SIP digest authentication, and therefore, the one modeled in our study.

4.1 Test bed configuration

Our experimental test bed consists of three main components:

- SIP Proxy: we used OpenSER 1.3.2 [19] as our SIP proxy. OpenSER is a mature and stable open source SIP proxy optimized for high performance. OpenSER was configured with minimal functionality (only required modules were enabled). A single stateless proxy was used in our tests. OpenSER was installed in a server with 8 2-GHz Quad-Core AMD Opteron processors, 16 GB of memory, 1 Gbit Ethernet card and running 2.6.24 Linux Kernel (Ubuntu 8.04.2).

- Remote database: MySQL 5.0.51a [2] was used as the database software for storing OpenSER's configuration data and subscriber's information (including authentication credentials). A default configuration was used in our test (no database optimizations were used). The database was populated with 10,000 subscribers, all belonging to a single SIP domain. MySQL was installed in a server with the same hardware and operating system as the server used for the proxy.

- User Agents (UAs): to simulate the SIP workload generated by the UAs, we used SIPp 3.1 [3], an open source test tool and traffic generator for the SIP protocol. A total of 24 SIPp instances were used to generate the workload, divided in two groups: 12 UA clients (UAC) and 12 UA servers (UAS). The number of SIPp instances was determined based on the hardware resources available. A total of 7 servers were used for running the SIPp instances (multiple instances per server). The default SIPp scenarios were modified to support digest authentication, according to the call dialog show in Figure 2 (only INVITE messages are authenticated). The SIPp servers had similar hardware and operating system as the servers used for the proxy and database.

The above components communicated using a dedicated Gigabit Ethernet network. To simulate the network latency between the proxy and the database, we used the Linux traffic control tool [1] with the network emulation module (netem). A network latency value of 30 ms was used in our experiments. This is a conservative value to a centralized location on the continent considering that the typical coast-to-coast network latency in the United States is between 80 to 100 ms.

Experiments were executed using a combination of Bash and Perl scripts. Additional open source tools such as iftop, pidstat, mpstat and vmstat were used to capture the required metrics in each test.

4.2 Implementing batch requests in OpenSER

In our study, we evaluated the use of batch requests to improve proxy performance. The basic idea behind this approach is to avoid individual requests to the database. Instead, requests are temporarily stored at the proxy and sent together as one multi-condition request. For example, a batch request of size 2 uses the following SQL query syntax:

```
"Select ha1 from subscriber where
 username='00033' OR username='002459'"
```

The previous SQL query returns the authentication credentials of two subscribers using a single round trip to the database. Using this approach, we can reduce the impact of the bandwidth overhead associated with individual requests. More details about this mechanism are presented in section 5.3

OpenSER does not support batch requests. Therefore, it was necessary to modify OpenSER to add this mechanism. Our code was added to OpenSER's authentication module. However, it was also necessary to modify OpenSER's core components to enable batch requests. Our experimental code is available to the community as a software patch for OpenSER version 1.3.2 at `http://www.cc.gatech.edu/~idacosta/proxy_batch.html`

4.3 Methodology

Three configurations were evaluated in our tests: the use of multiple processes, the use of batch requests and the combination of both mechanisms. To evaluate each configuration, we focused on the following metrics: call throughput at the proxy, message retransmissions, failed calls, bandwidth between the proxy and the database and CPU utilization for the proxy and database.

Call throughput corresponds to the number of successful calls per second (cps) measured in each time period (5 s). The maximum call throughput was determined as the highest load where the number of failed calls remained under 1% of the total load (maximum usable throughput with 1% failure tolerance). We selected this bound as it is commonly applied in more traditional telephony networks. Message retransmission refers to the number of SIP messages being retransmitted due to the expiration of timers in the SIPp instances. We used the default retransmission time recommended by the SIP RFC: 500 ms. Failed calls corresponds to the number of unsuccessful calls in the last period measured by the SIPp UASs. The two main reasons for call failures in our tests were unexpected messages (messages out of order) and maximum number of retransmissions (maximum number of UDP retransmission attempts has been reached). We used the default values in SIPp for UDP retransmissions: 5 for INVITE transactions and 7 for others.

Each test was run for 10 minutes. During this period, the SIPp instances generated an increasing SIP workload. The workload was increased every 5 seconds by a constant amount. The amount of increase was adjusted according to the configuration evaluated (depending on the number of processes running on the proxy), ranging from 12 to 180 cps (to avoid saturation of the proxy too quickly). During each test, performance statistics were collected by the UASs. Additional data (i.e., bandwidth and CPU utilization) was collected using the tools described at the end of subsection 4.1.

To ensure the validity of our results, each test was repeated 10 times. Average values were used in our analysis and a 95 % confidence interval is provided. Approximately 600 unique tests were executed during our study.

In the next section we present the results of our tests for each configuration and our analysis.

5. ANALYSIS OF THROUGHPUT ENHANCEMENT TECHNIQUES

5.1 Standard configuration

Our first test evaluated the performance of two stateless SIP proxy configurations: non-authentication (base case) and digest authentication with remote database (RTT=30 ms). Both configurations used 4 child processes (default OpenSER value). The results are presented in Figure 3.

For the non-authentication configuration, the maximum throughput registered was approximately 20,000 cps. However, this was

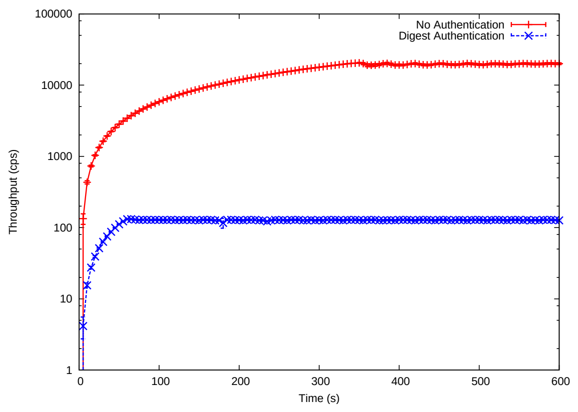

Figure 3: Proxy's throughput with and without digest authentication. The network latency (30 ms) between the proxy and the database significantly reduces proxy's performance

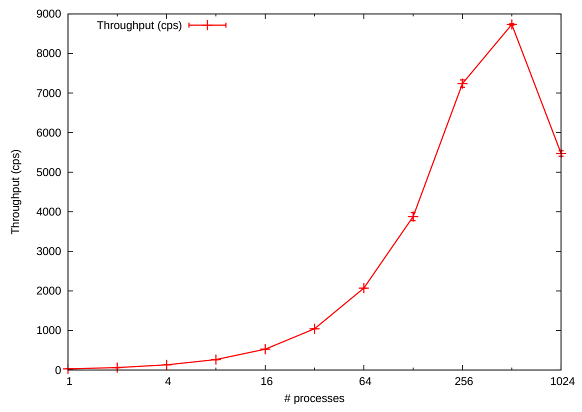

Figure 4: Proxy's maximum usable throughput (<1% failed calls) for different number of processes. Proxy's performance improves with higher number of processes. However, performance drops sharply when 1024 processes are used.

not the maximum throughput that the proxy could handle. At 20,000 cps the SIPp instances were unable to continue increasing the workload. In other words, the SIPp instances reached saturation before the proxy. At 20,000 cps the proxy had a CPU utilization of 20%. Therefore, we can estimate that the maximum throughput that our proxy can handle is significantly larger.

For the digest authentication configuration, the maximum throughput registered was approximately 130 cps. This value represents a drop in performance of almost three orders of magnitude, but is logically correct as it closely matches the estimation $1000\,ms/30\,ms\,delay = 33 * 4\,processes$. This drop in performance also means that the proxy and database are heavily underutilized (less than 1% CPU utilization).

The main reason for this significant drop in performance is the network latency between the proxy and the database. When digest authentication is not used, the proxy routes messages very quickly (less than 50 μs per message). However, when digest authentication is enabled, the proxy needs to contact the database to get the subscriber's authentication credentials to verify the subscriber's request (AUTH box in Figure 2). This operation adds approximately 30 ms to the total session setup. This additional time slows down the processing of messages in the proxy, because each OpenSER process routes messages serially. Each time an OpenSER process needs to contact the database, it stops processing other messages and waits for the database's response (blocking call).

One approach to reduce the effect of blocking calls and network latency, is to increase the number of parallel operations performed by the proxy. In other words, increase the number of OpenSER child processes. This approach is often recommended by OpenSER developers to improve performance. In the next section we present the evaluation of this option.

5.2 Improving performance with multiple processes

To evaluate the use of multiple processes to improve performance, we measured the proxy's throughput over a range: from 2 to 1024 processes, increasing our interval by a factor of 2. Figure 4 shows the results of our experiments. As expected, the proxy's throughput improves as the number of processes increases. Intuitively, this improvement is due to the increased probability that a non-blocked process will be available when a new request arrives at the proxy.

It is important to notice that the throughput values in Figure 4 represent the *maximum usable throughput* (1% call failure tolerance) measured in our tests. In most cases, this value is equivalent to the maximum throughput handled by the proxy. However, in certain configurations, the maximum throughput has a high percentage of call failures ($>> 1\%$), which makes this value impractical in production environments.

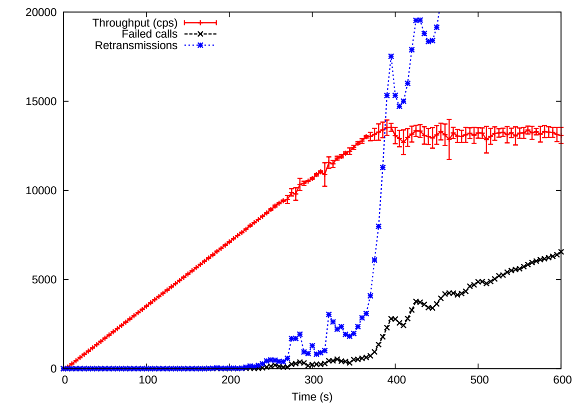

Figure 5: Proxy's throughput for the 512 processes configuration. The maximum throughput that the proxy can handle is limited by the database maximum throughput.

Figure 4 depicts the maximum usable throughput for 512 processes: almost 9,000 cps. This value represents a considerable improvement in performance when compared with the value obtained in the previous section (130 cps). Our results also show that at 1024 processes, the maximum usable throughput decreased markedly to approximately 5,500 cps. Even though the performance improvement is significant, it still falls short when compared to the proxy's throughput with no authentication (approximately 20,000 cps). To understand why the proxy can not handle higher throughput, we analyzed in more detail the differences between configurations using 512 and 1024 processes.

Figure 5 provides a more in-depth characterization of the behavior of 512 processes. As previously mentioned, traffic is gradually increased with time to create performance profiles. Two critical observations can be made from this configuration. First, the maximum throughput is approximately 13,000 cps; however, the corresponding rate of failed calls is intolerably high (almost 21% of

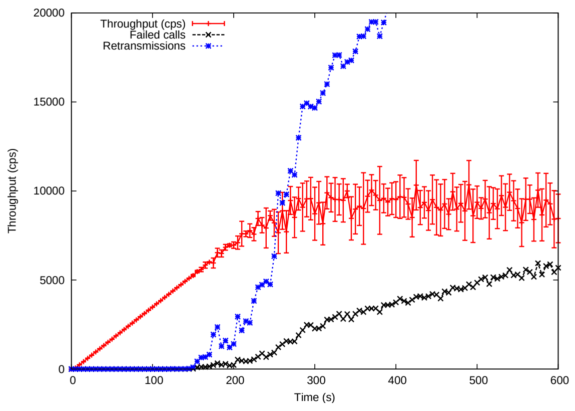

Figure 6: Proxy's throughput for 1024 processes configuration. Trashing in the proxy due to the high number of processes degrades proxy's performance

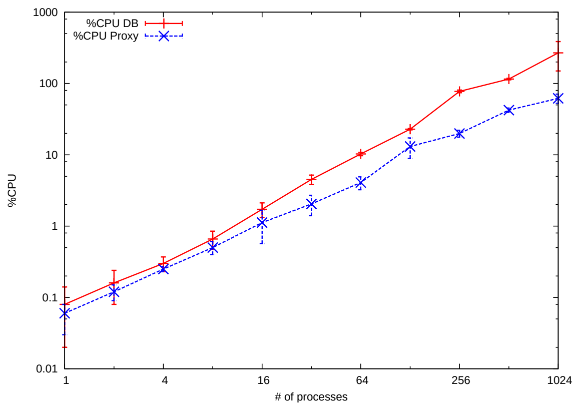

Figure 7: Proxy and database CPU utilization for different number of processes. Around 512, the database CPU utilization is more than 100 % and the database becomes a bottleneck for proxy's performance

the total throughput). Second, message retransmissions begin before reaching the maximum throughput and grow extremely rapidly once it is reached. The number of failed calls also grows faster once the maximum throughput is reached. Typically, the number of retransmissions and failed calls increases because the proxy can not process the messages fast enough (the proxy is too busy). However, the proxy's CPU utilization at the maximum throughput was approximately 40%. Figure 7 provides additional context to clarify this discrepancy. Specifically, around 512 processes the database application is using more than 100 % CPU utilization (multiprocessor machine). At 512 processes, the database application is using almost 120% CPU utilization. At this point the database can not process a higher rate of requests from the proxy, and as a result, the database becomes the performance bottleneck.

Based on our results and assuming the use of a faster database (i.e., an in-memory database), we could continue increasing the number of processes to improve the proxy's throughput. For example, the proxy could theoretically handle a throughput of almost 35,000 cps with a 100% CPU utilization (13,000 cps uses 40% CPU utilization). At that point, the proxy's CPU will be the bottleneck. However, Figure 6 contradicts this assertion. Note that retransmissions and call failures happen earlier (at a lower traffic rate) during

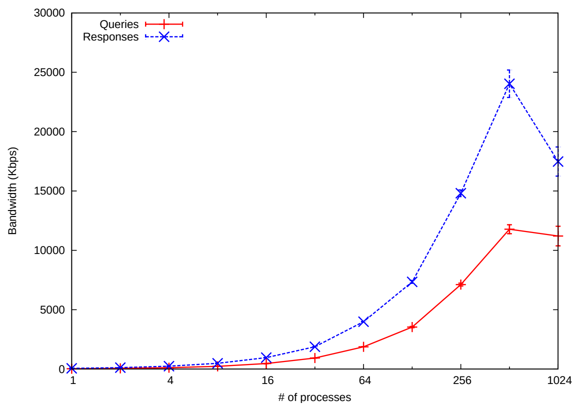

Figure 8: Bandwidth between the proxy and the database for different number of processes. A considerable amount of bandwidth is required for high number of processes

the tests on 1024 process when compared with the 512 processes case. The maximum throughput for 1024 processes is also less than the 512 process case, indicating that the database is not the only cause for the drop in performance in this configuration. The reason for this behavior is that the high number of processes begins to degrade the proxy's performance due to thrashing (an elevated number of page faults and context switches). As a result, we can conclude that only increasing the number of processes is not enough to maximize throughput.

The use of simple parallelization techniques also causes bandwidth usage between the proxy and the database to become a concern. Figure 8 shows the bandwidth for the queries-to and responses-from the database. The response bandwidth is higher than the query bandwidth because the queries have smaller size than the corresponding responses (authentication credentials). Note that bandwidth grows linearly with the number of processes. For example, 512 processes require a bandwidth of 12 Mbps for queries and 24 Mbps for responses. These bandwidth values can be prohibitive for some scenarios, specially if the database and the proxy communicate across the public Internet. The bandwidth between the proxy and the database may therefore limit the number of processes that can be used at the proxy.

The network and application layer overheads are the main reason for the high bandwidth requirements. A single query has a packet length of 126 bytes, with 56 bytes of payload (56% overhead). The corresponding response has a packet length of 247 bytes with 146 bytes of payload (41% overhead). Therefore, almost half of the traffic exchanged between the proxy and the database corresponds to network and application layer headers and control packets. The situation is made worse if we consider the use of security protocols such as IPsec or SSL to protect this communication channel.

Finally, we note that OpenSER creates an independent TCP connection to the database for each child process it spawns. This approach is not very efficient if a high number of processes are used (i.e., a TCP port per process needs to be allocated). A more efficient approach could be to share a pool of TCP connections to the database among all the processes. However, we did not attempt to retrofit this codebase to include this optimization.

5.3 Improving performance with batch requests

The second approach to improve proxy performance is the use of request batching. The idea is to reduce the number of times the proxy needs to access the database so as to avoid the impact

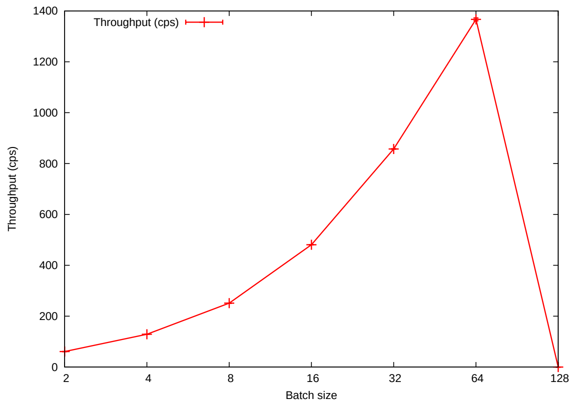

Figure 9: Proxy's maximum usable throughput for different batch sizes using a single process. The performance improvement is lower when compared with multiple processes.

of the high round trip time between the proxy and the database (the most expensive step of the authentication process). Instead of sending single queries to the database, the proxy holds requests in a queue of size n. Once the queue is full,[1] the proxy sends a batch query to the database to retrieve the corresponding n authentication credentials in one single round trip.

Based on the above, the time a batch query takes to retrieve n credentials should be less than the total time n single queries take to retrieve n credentials, as the following inequality shows:

$$RTT_{pd} + t_{batch} < n \times (t_{db} + RTT_{pd}) \quad (1)$$

where RTT_{pd} is the network latency between the proxy and the database, t_{batch} is the query execution time of a batch query of size n, and t_{db} is the query execution time of a single query (query execution time corresponds to the time the database takes to execute a query).

Assuming that $t_{batch} = n \times t_{db}$ (worst case scenario), the impact of network latency is reduced by a factor n when a batch query is used instead of single queries. Using this result, we can estimate the time t_{sq} each query in the batch takes:

$$t_{sq} = \frac{RTT_{pd} + t_{batch}}{n} \quad (2)$$

Assuming $t_{batch} = n \times t_{db}$ we have:

$$t_{sq} = \frac{RTT_{pd}}{n} + t_{db} \quad (3)$$

Equation 3 shows that the impact of the network latency is effectively amortized among the n queries in the batch.

To evaluate the effectiveness of using batch requests, we measured the proxy's maximum usable throughput for different batch sizes: from 2 to 128, increasing the batch size by a factor of 2 in each step. Figure 9 shows the results of these tests. The maximum usable throughput occurred with a batch size of 64: almost 1,400 cps. This performance improvement is small when compared with the parallel execution approach, but note that these tests where executed with a single process. It can therefore already be concluded

[1]Because we have relatively high amounts of traffic, we are not worried about starving requests by forcing them to wait for the queue to fill. A real deployment could avoid such an issue by incorporating a timer, which would cause the batch to be sent even if the queue was not full after some period.

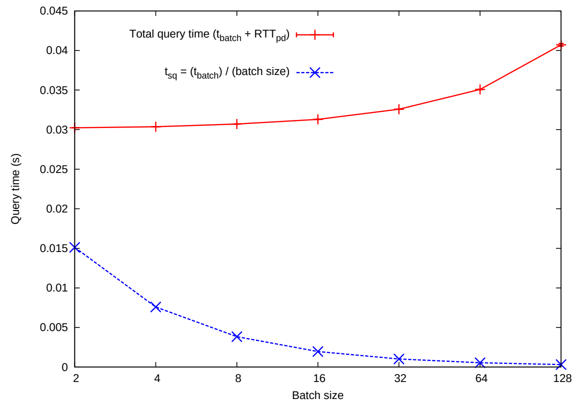

Figure 10: Total query time ($t_{batch} + RTT_{pd}$) and $t_{sq} = \frac{RTT_{pd}+t_{batch}}{n}$ for different batch sizes. Using batch sizes larger than 64 has little effect on improving performance

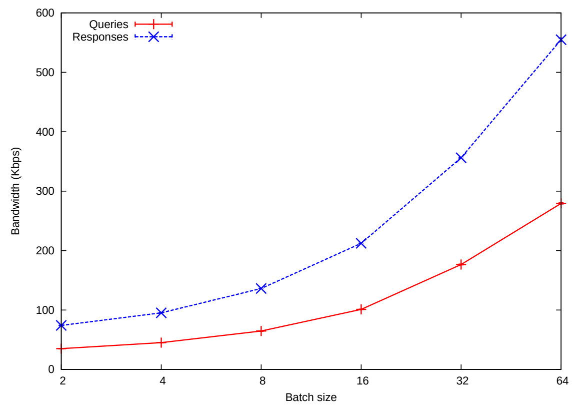

Figure 11: Bandwidth between the proxy and the database for different batch sizes. Batch requests have lower bandwidth requirements than multiple processes

that batch requests alone are not enough to maximize proxy's performance.

Figure 9 also shows that the maximum usable throughput with a batch size of 128 is close to 0 cps. With this batch size, the number of retransmissions and failed calls is extremely high almost immediately after the start of the experiments. These results provide us with an estimation of the maximum batch size that can be used in our scenario.

Due to the lower throughput values obtained using only batch requests, the CPU utilization in both, the CPU and the database, was less than 10 % for all the tests.

To confirm the validity of Equation 1, we measured the total query time ($t_{batch}+RTT_{pd}$) for different batch sizes. Figure 10 depicts how the batch query time increases with the batch size. However the rate of increase is much lower than using single queries. For example, a single query takes approximately 30.2 ms, therefore, 32 queries will take approximately 964.6 ms. Instead, a batch query of size 32 takes approximately 32.581 ms. Figure 10 also shows how the network latency is amortized among the queries in the batch (Equation 3). It is important to notice that after a batch size of 64, the curve begins to flatten out. This behavior indicates that larger batch sizes will have little effect on reducing the impact of network latency.

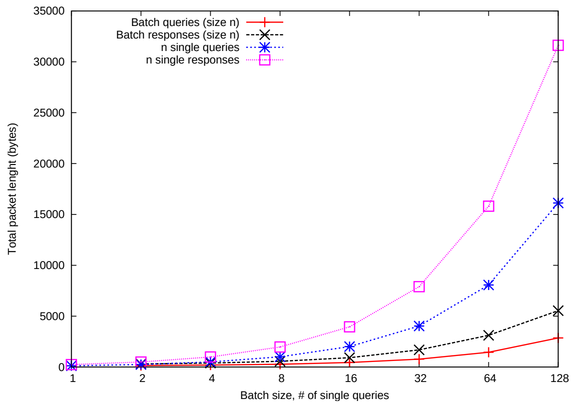

Figure 12: Total packet length for batch queries and single queries. Batch queries have better payload efficiency than the equivalent number of single queries

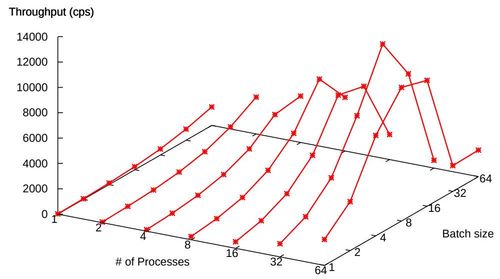

Figure 13: Proxy's maximum usable throughput for different combinations of number of processes and batch sizes. The best performance is achieved with the 32 processes with batch size of 16

Figure 11 shows that the use of batch requests also reduces the bandwidth requirements between the proxy and the database. For example, a proxy with 8 processes has a maximum throughput of 265 cps and requires 232 and 482 Kbps for query and response bandwidth respectively. The same proxy with a single process and a batch size of 8 has a maximum throughput of 251 cps and requires 64.70 and 136.20 Kbps for query and response traffic respectively. Batching therefore required 71.8% less bandwidth than recommended multiple processes technique for the same throughput.

Batch requests require lower bandwidth because they have better payload efficiency (payload/total packet size) than multiple processes. Figure 12 shows how batch requests have a lower total packet length than the similar number of single queries. For example, 8 single queries will use a total packet length of 1008 and 1976 bytes for the query and the response, respectively. A batch query of size 8, will require 277 and 569 bytes instead. A batch query of size 8 has a payload of 207 bytes and 450 bytes for the query and the response packets (25% and 21% overhead); 8 single queries have a payload of 448 and 1168 bytes (56% and 41% overhead).

A tradeoff with the use of batch requests is added latency in session establishment time associated with the queuing period. In particular, the first request to arrive to the queue will have the longest delay (worst case). This request will have to wait for the queue to fill up first in order to continue with authentication procedures. However, this additional latency is virtually unnoticeable to users as session setups in telephony are on the order of seconds [10].

Given all of these inputs, we now attempt to select an appropriate batch size. In general, this will be a scenario dependent value. Equation 3 indicates that using larger batch sizes is the best strategy to reduce the effect of network latency. However, increasing the batch size will also increase the execution time on the database (t_{db}), as shown in Figure 10. After a certain point, increasing the batch size will no longer improve performance, especially if the database is not optimized for processing large batch requests. Our results show that for our scenario, batch sizes larger than 64 will not help to improve performance; on the contrary, they will degrade it (see Figure 9).

There are also other factors to consider when choosing the batch size. Using larger batch sizes will also increase the delay introduced to the session setup, and produce unnecessary retransmissions and failed calls if the queue is not filled up fast enough. In general, the queue has to be filled up before the retransmission time of the first request in the queue expires (default retransmission time is 500 ms). Using larger batch sizes will also increase the length of the queries and responses packets. Once the packet length is bigger than the network MTU (Maximum Transmission Unit), fragmentation will occur. Fragmentation decreases the payload efficiency because additional control information is required (i.e., more headers). Even worse, fragmentation will also increase the likelihood of packet loss, and therefore, excessive retransmissions (retransmission of all the fragments). This factor is especially important in Internet scenarios. In Figure 12, a batch request of size 128 requires 2 fragments for the queries, and 4 fragments for the responses (plus additional TCP control packets).

A final factor to consider is the proxy's design and implementation. We found several problems and error messages with batch sizes of 128 or larger. We concluded that OpenSER's design is one of the main reasons for the performance degradation when a batch size of 128 was used.

5.4 Hybrid approach: combining multiple processes with batch requests

The use of multiple processes is an effective way to improve throughput. However, it is not efficient in terms of the bandwidth used between the proxy and the database. Request batching, alternatively, is not effective for maximizing throughput, but does improve performance with dramatic reductions in communications and application-layer overhead. The complimentary nature of these two techniques makes their combination a logical next step. To verify that a hybrid of these two mechanisms can further improve performance, we tested the combination of different number of processes (2, 4, 8, 16, 32, and 64) with different batch sizes (2, 4, 8, 16, 32, and 64) - a total of 36 tests.

Figure 13 shows the maximum usable throughput values for each configuration. The maximum usable throughput is reached with the combination of 32 processes with a batch size of 16 (32p-16b): approximately 12,100 cps. This value represents a 34% improvement over the maximum usable throughput obtained with multiple processes (9,000 cps for 512 processes).

Figure 13 also shows that configurations with higher number of processes or larger batch sizes than 32p-16b (i.e., 32p-32p or 64p-32p) exhibit poorer performance than the 32p-16b configuration. These other configurations are able to handle approximately the

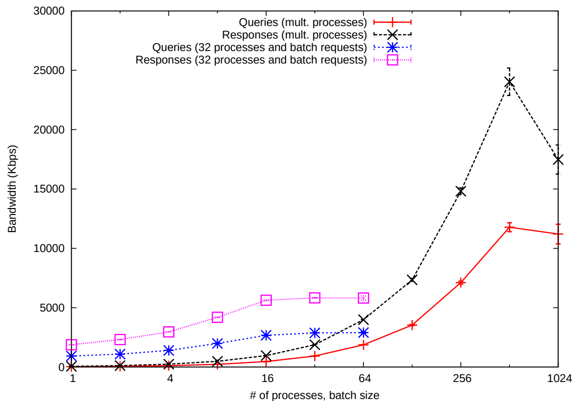

Figure 14: Bandwidth between the proxy and the database for different number of processes and for 32 processes with different batch sizes. The combination of multiple processes with batching required significantly less bandwidth than using multiple processes only.

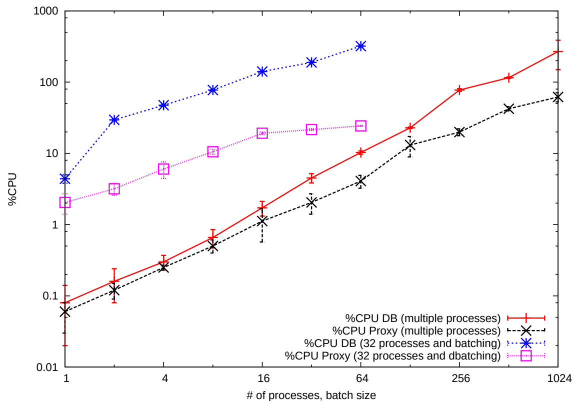

Figure 15: Proxy and database CPU utilization for different number of processes and for 32 processes with different batch sizes. The combination of multiple processes with batching requires less proxy CPU time than multiple processes

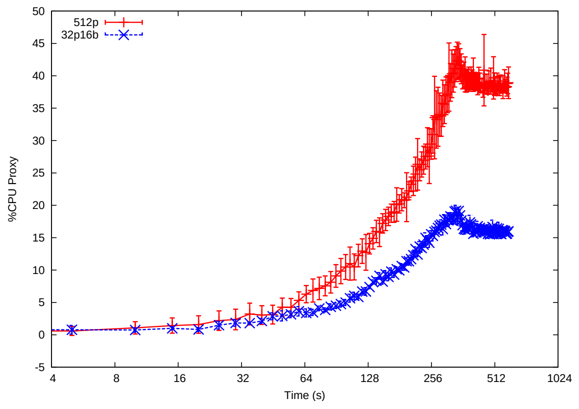

Figure 16: Proxy CPU utilization for 512 processes and for 32 processes with batch size of 16. Both configurations handle approximately the same maximum throughput but the hybrid approach requires less CPU time

same maximum throughput than 32p-16b but also have more retransmissions and call failures, which degrades their usable throughput. Such degradation is caused by a combination of database saturation and the operation issues associated with OpenSER and SIPp. If a faster database were used (e.g., in-memory), these configurations would likely perform better than 32p-16b; however, our characterization is valuable as it describes how OpenSER is currently configured.

Configurations such as 16p-32b also perform worse than the 32p-16b scenario. While these configurations appear to be equivalent, in practice they are not. The reason is that the use of a higher number of processes is more effective to maximize throughput than using higher batch sizes.

Figure 14 presents the measured bandwidth between the proxy and the database for parallel execution and for the 32 processes hybrid approach. The figure shows a significant difference between the bandwidth used by multiple processes and the bandwidth used by the hybrid approach. For example, the best hybrid configuration tested (32p-16b) requires 2,676 Kbps for queries to the database and 5,625 Kbps for responses from the database. In contrast, the best multiple processes configuration (512p) requires 11,780 Kbps for queries to the database and 24,030 Kbps for the responses from the database. This is an improvement of 77.3% and 76.6% for requests and responses, respectively. Therefore, the 32p-16b configuration requires 4 times less bandwidth than the 512p configuration. The main reason for this significant difference is that batch requests have better payload efficiency than single queries (see Section 5.3).

Finally, we compare the CPU utilization in the proxy and the database for the hybrid approach. Figure 15 compares the CPU utilization for different number of processes against the CPU utilization for 32 processes with different batch sizes. In the database's case, the figure shows that there is not a significant difference between using multiple processes or the hybrid approach; both techniques saturate the database (more than 100% CPU utilization). However, in the proxy's case, the hybrid approach uses less CPU time than multiple processes. For example, for the 32p-16 case, the CPU utilization is almost 20%. For 512 processes, the proxy CPU utilization is almost 42%. Figure 16 depicts this difference in greater detail.

To understand why the 512p configuration requires higher proxy CPU utilization, we compare Figure 5 and Figure 17. Both configurations handled approximately the same maximum throughput, however, the 512p configuration has more retransmissions and call failures earlier than the 32p-16b configuration. As a result, the proxy has to do extra processing when the 512p configuration is used. In addition, the use of a high number of processes increases the probability of more overhead (i.e., more context switching) in the proxy, consuming additional CPU time.

6. DISCUSSION

Table 1 shows the results of the best configuration of each technique: multiple processes (512 processes), batch requests (one process with batch size of 64), and the hybrid approach (32 processes with bath size of 16). These results demonstrate that the hybrid approach effectively combines the good properties of the other two approaches: improved throughput values and lower bandwidth between the proxy and the database.

Based on the above results, 32p-16b provides the best performance for a distributed proxy architecture with similar components. We note, however, the need to carefully interpret these numbers. While the results indicate that the hybrid approach can handle higher throughput than the multiple processes approach, we did not test

Configuration	Throughput	Bandwidth (Query/Response)	Proxy CPU	DB CPU
512 processes	9,000 cps	11,780 / 24,030 Kbps	42%	115%
1 process with batch size = 64	1366 cps	279 / 555 Kbps	6%	2%
32 processes with batch size = 16	12,100 cps	2,676 / 5,625 Kbps	20%	135%

Table 1: Best configuration for each technique

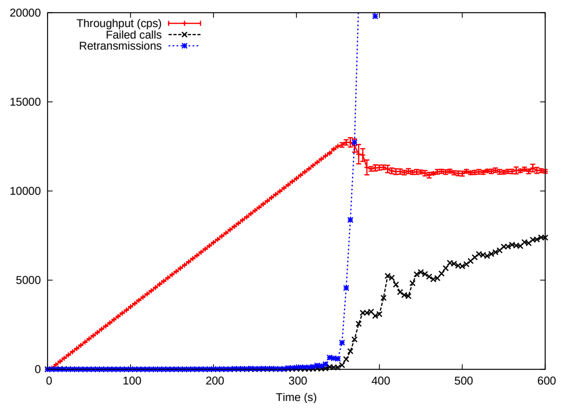

Figure 17: Throughput for 32 processes with batch size of 16. Notice the lower number of retransmissions and failed calls when compared with the 512 processes configuration (Figure 5)

every possible configuration for each technique. It may be possible that a different number of multiple processes (somewhere between 256 and 512) offers performance closer to that of the 32p-16b configuration. Such a configuration is still unlikely to improve over the hybrid approach due to the associated bandwidth overhead. It may also be possible that a different mix of multiple processes and batch size have even better performance. Accordingly, these results should be used as a guide and aid network administrators in their parameter setting. Both the 512p and 32p-16b configurations can handle approximately the same maximum throughput ($\approx$ 12,000 cps), which is bounded by the maximum database's throughput. The difference resides in the number of retransmissions and call failures that happen in each configuration, which results in different maximum usable throughput values. This difference can be observed comparing Figure 5 and Figure 17. As a result, *the hybrid approach can maximize throughput as effectively as running multiple processes but using lower number of processes.* The use of lower number of processes helps to reduce context switching overheads and thrashing situations.

In terms of bandwidth requirements, the hybrid approach is much more efficient than any configuration of multiple processes. As Figure 8 shows, configurations larger than 128p require considerable bandwidth between the proxy and the database. In conclusion, *for similar throughput values, a hybrid configuration will require much less bandwidth than a multiple processes configuration.*

CPU utilization in the database is approximately the same for both, the hybrid and multiple processes techniques. However, the CPU utilization in the proxy will be higher for multiple processes because of the context switching overhead. Therefore, *for comparable throughput values, the hybrid approach will require the same or less proxy CPU time than a multiple processes configuration*

As a tradeoff, the use of batch requests will increase the session setup time in low traffic scenarios. However, the delay introduced is unlikely to be noticed by humans. Additionally, using batch requests increase the packet length of the queries and responses between the proxy and the database. Larger packet sizes may require fragmentation, making this approach more susceptible to packet losses at very large batch sizes.

In general, designers and developers can use the following rule when applying the hybrid approach: *for comparable throughput values, increasing the number of processes improves throughput to a point, but the use of request batching will be required to reduce the overhead associated with this technique in order to truly maximize the usable throughput of the proxy.*

It is not a coincidence that the to number of processes in the multiple processes approach is equal to the product of the number of processes and its batch size in the hybrid approach (512=32*16). If we look again at Figures 4 and 9, we can see that for batch sizes lower than 16, approximately the same throughput is handled using n multiple processes, or using one process with a batch size of n. Therefore, *a configuration using n multiple processes will handle approximately the same throughput as a configuration with p processes with batch size=q, where p and q are factors of n and* $q \leq 16$.

When using the hybrid approach in a production environment, we recommend having batch requests enabled all the time. As mentioned before, the extra-delay introduced by batch requests is unlikely to be noticed by humans. Also, the use of timers will avoid unnecessary retransmissions due to timeouts when the call load is low. In this way, the system will be always prepared for events where the call load increases unexpectedly (i.e., flash crowds, distributed denial of service attacks and emergency situations).

In future work, we plan to develop a mathematical model for the relationship between the number of processes and the batch sizes. The idea is to determine the optimal values according to the characteristics of the scenario of interest. We are also interested in investigating alternatives to avoid the bottlenecks associated with the database.

7. CONCLUSION

SIP proxies are often distributed across a wide geographic area in order to minimize latency between themselves and clients. However, the supporting authentication services are often centrally located, potentially leading to the degradation of call throughput due to network latency. In this paper, we investigated two schemes to improve proxy throughput in such an architecture. First, we demonstrated that the commonly recommended approach, parallel execution, improved performance but quickly caused the rate of call failure to rise. We then implemented a request batching mechanism that reduced the bandwidth overhead associated with previous approach, but failed to reach a sufficiently high throughput. Finally, we created a hybrid of these two schemes that improved throughput by more than 34%, reduced bandwidth overhead by more than 75% over the best parallel execution method and maintained loss rates below 1%. Future work will investigate techniques to overcome the database bottleneck and further reduce request latency.

8. REFERENCES

[1] Introduction to linux traffic control. http://linux-ip.net/articles/traffic-control-howto/intro.html.

[2] Mysql ab - the world's most popular open source database. http://www.mysql.com/.

[3] SIPp: traffic generator for the SIP protocol. `http://sipp.sourceforge.net/`.

[4] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[5] V. Balasubramaniyan, A. Acharya, M. Ahamad, M. Srivatsa, I. Dacosta, and C. Wright. SERvartuka: Dynamic Distribution of State to Improve SIP Server Scalability. *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 562–572, June 2008.

[6] J. Black, M. Cochran, and T. Highland. A Study of the MD5 Attacks: Insights and Improvements. In *Fast Software Encryption*, 2006.

[7] D. Boneh and H. Shacham. Improving SSL Handshake Performance via Batching. In *RSA '2001, Lecture Notes in Computer Science, Vol. 2020, Springer-Verlag*, pages pp. 28–43, 2001.

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

[9] M. Cortes, J. O. Esteban, and H. Jun. Towards Stateless Core: Improving SIP Proxy Scalability. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, pages 1–6, 2006.

[10] T. Eyers and H. Schulzrinne. Predicting Internet Telephony Call Setup Delay. In *Proc. 1st IP-Telephony Wksp*, 2000.

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

[12] L. Guernsey. Keeping the Lifelines Open. *The New York Times*, http://www.nytimes.com/2001/09/20/technology/circuits/20INFR.html, September 2001.

[13] J. Janak. Sip proxy server effectiveness. *Master's Thesis, Department of Computer Science, Czech Technical University, Prague, Czech*, 2003.

[14] J. Kim, A. Biryukov, B. Preneel, and S. Hong. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1. In *Security and Cryptography for Networks*, 2006.

[15] R. Lemos. Cyber attacks disrupt Kyrgyzstan's networks . securityfocus. http://www.securityfocus.com/brief/896, January 2009.

[16] H. Liu. Applying Queuing Theory to Optimizing the Performance of Enterprise Software Applications. In *CMG-CONFERENCE-*, volume 1, page 457. Computer Measurement Group; 1997, 2006.

[17] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP server performance, 2007.

[18] K. Ono and H. Schulzrinne. One Server Per City: Using TCP for Very Large SIP Servers. In *The 2nd LNCS Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, Heidelberg, Germany, jul 2008.

[19] OpenSER. OpenSER - the Open Source SIP Server. `http://www.openser.org/`.

[20] M. Richtel. Inauguration Crowd Will Test Cellphone Networks. *The New York Times*, http://www.nytimes.com/2009/01/19/technology/19cell.html, 2009.

[21] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261: SIP: Session Initiation Protocol, 2002.

[22] S. Salsano, L. Veltri, and D. Papalilo. SIP security issues: the SIP authentication procedure and its processing load. *Network, IEEE*, 16(6):38–44, 2002.

[23] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone-Benchmarking SIP Server Performance. 2002.

[24] C. Shen, H. Schulzrinne, and E. Nahum. Session Initiation Protocol (SIP) Server Overload Control: Design and Evaluation. In *IPTComm 2008*, pages 149–173.

[25] K. Singh and H. Schulzrinne. Failover and load sharing in SIP telephony. *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), Philadelphia, PA, July*, 2005.

[26] K. Singh, H. Schulzrinne, and J. Lennox. SIP Server Scalability, 2005.

[27] K. N. Singh. *Reliable, Scalable and Interoperable Internet Telephony*. PhD thesis, COLUMBIA UNIVERSITY, 2006.

[28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[29] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of EUROCRYPT*, 2005.