Explaining the Impact of Network Transport Protocols on SIP Proxy Performance

Kaushik Kumar Ram, Ian C. Fedeli, Alan L. Cox, and Scott Rixner
{kaushik,ifedeli,alc,rixner}@rice.edu
Rice University

Abstract

This paper characterizes the impact that the use of UDP versus TCP has on the performance and scalability of the OpenSER SIP proxy server. The Session Initiation Protocol (SIP) is an application-layer signaling protocol that is widely used for establishing Voice-over-IP (VoIP) phone calls. SIP can utilize a variety of transport protocols, including UDP and TCP. Despite the advantages of TCP, such as reliable delivery and congestion control, the common practice is to use UDP. This is a result of the belief that UDP's lower processor and network overhead results in improved performance and scalability of SIP services.

This paper argues against this conventional wisdom. This paper shows that the principal reasons for OpenSER's poor performance using TCP are caused by the server's design, and not the low-level performance of UDP versus TCP. Specifically, OpenSER's architecture for handling concurrent calls is responsible for most of the difference. Moreover, once these issues are addressed, OpenSER's performance using TCP is much more competitive with its performance using UDP.

1 Introduction

Despite the increasing deployment of Voice-over-IP (VoIP) services, particularly those based upon the IETF's Session Initiation Protocol (SIP), very little information has been published about the design and performance of SIP servers. In essence, a SIP server acts as the "switchboard operator" establishing the connection between the SIP client phones participating in a call. As such, it is a potential bottleneck. This paper addresses a key gap in the literature about SIP servers. It examines the relative importance of two factors that impact a SIP server's throughput and scalability: (1) the server's architecture for handling concurrent calls and (2) its method of communication with a client phone. In particular, this paper refutes previous claims that the method of communication is the more important factor [11].

A SIP server can utilize a variety of transport protocols, including UDP and TCP, for implementing communication between itself and phones. Despite the advantages of TCP, such as reliable delivery and congestion control, UDP is most widely used in practice. UDP is used instead of TCP because of the belief that the additional complexity of TCP connection management results in significantly lower throughput and scalability.

This paper shows that the principal reasons for OpenSER's poor performance using TCP are, in fact, caused by the server's design, not the low-level overhead differences between UDP and TCP. A higher-level difference between TCP and UDP does, however, play a role. Specifically, OpenSER's architecture for handling concurrent calls is better suited to a connection-less protocol, such as UDP, than a connection-oriented protocol, such as TCP.

OpenSER uses a process-based architecture for handling concurrent calls. Under UDP, each of these processes is identical in function. Roughly speaking, each one executes an event loop in which it receives a message from a phone or server, computes the next hop in the route that the message should follow, and forwards the message to the phone or server that is the next hop. Moreover, any of these *worker* processes can receive a message from any source and send it to any destination.

In contrast, under TCP, the architecture is asymmetric. Like UDP, there are many worker processes. Unlike UDP, there is also a single *supervisor* process that is dedicated to connection management. This supervisor process accepts all connections on behalf of the server and assigns "ownership" of each connection to a worker process. As the owner, that worker process is solely responsible for receiving messages from that connection.

Although the owner is the only process to receive messages from a connection, a connection may be written to by different sending processes. This is realized by sharing

This work was supported in part by the National Science Foundation under Grant No. CCF-0546140 and by gifts from Advanced Micro Devices.

connections among the processes. Specifically, the supervisor maintains a socket file descriptor for every connection held by the server. When a worker process must forward a message on a connection, it requests its own file descriptor for the connection through interprocess communication (IPC) to the supervisor. After the worker has forwarded the message on the connection, it closes its file descriptor.

This paper shows that the principal reasons for OpenSER's low performance using TCP are a result of this architecture. Specifically, two key design decisions cause the bulk of the performance loss. First, the frequency and overhead of IPC for requesting file descriptors for connections have a profound effect on performance. Second, the strategy for managing idle connections is inefficient and time consuming. Moreover, once these issues are addressed, OpenSER's performance using TCP is competitive with its performance using UDP. Specifically, OpenSER's performance using TCP increases from 13-51% to 50-78% of the performance using UDP.

The rest of this paper is organized as follows. The next section presents an overview of SIP. Section 3 then describes OpenSER's architecture and, in particular, the differences in how TCP and UDP are handled. Section 4 describes the experimental methodology that is used to characterize OpenSER's performance. Section 5 presents a characterization of OpenSER's performance using TCP and UDP, and Section 6 discusses the implications of these findings. Finally, Section 7 discusses related work, and Section 8 concludes the paper.

2 Session Initiation Protocol (SIP)

In VoIP distinct protocols are used for establishing connections between phones and carrying the voice data. The two major competing standards for establishing VoIP calls are the Session Initiation Protocol (SIP) [14] (IETF standard) and H.323 [6] (ITU Standard). Once the VoIP call is established using one of these protocols, the actual voice data transfer is usually handled directly between phones over a protocol such as the Real-time Transport Protocol (RTP) [15].

This paper considers SIP, which is a message-based protocol that establishes communication between phones via a sequence of SIP proxy and redirections servers. These proxy and redirection servers use a SIP location service in order to route SIP messages. Phones must *register* with a SIP registrar in order to update the SIP location service appropriately. A caller then identifies a callee using a SIP uniform resource identifier (URI) of the form `sip:username@domain`—for example, `sip:batman@gotham.gov`. To place a call, the caller will generate a SIP message and send it to a SIP proxy or redirection server that it knows about (presumably in the caller's domain). If the callee is locally registered with the server, it can then forward the message directly. If the callee is not locally registered, the server can either act as a proxy by forwarding the message to another server or remove itself from the transaction by redirecting the caller to a different server. Assuming the callee is registered, the message will ultimately reach a SIP server that knows the location of the callee. That server will send the message directly to the callee. The caller and callee will then use the SIP protocol, via the same sequence of SIP proxy servers that were used to route the original message, to establish the call so that they can begin the communication.

The sequence of SIP messages between the caller and callee are called a transaction. There are three distinct types of SIP transactions involved in VoIP calls: register, invite, and bye. A register transaction is used to register a phone with a SIP registrar, as previously discussed. An invite transaction is used to initiate a phone call and a bye transaction is used to end a phone call.

SIP proxy servers form the backbone of the SIP infrastructure, as they handle the routing and delivery of SIP messages within a transaction. Such a proxy server can either be stateful or stateless. A stateless proxy server does not keep track of any SIP messages involved in a transaction and leaves the burden of communication reliability on the caller and callee. In contrast, a stateful proxy server keeps track of all messages within a transaction and attempts to retry messages until the transaction is complete or times out.

SIP proxy servers deal primarily with invite and bye transactions between phones. An invite transaction via a SIP proxy server consists of the following steps:

1. The caller sends an INVITE message to the proxy in order to initiate a phone call.
2. A stateful proxy responds to the INVITE message from the caller with a TRYING message, indicating that the message has been received. A stateless proxy skips this step entirely.
3. The proxy forwards the INVITE message to the callee (or another SIP server).
4. The callee responds to the INVITE message from the proxy with a RINGING message, indicating that the callee's phone is ringing.
5. The proxy forwards the RINGING message to the caller.
6. When the phone is "picked up", the callee sends an OK message to the proxy.
7. The proxy forwards the OK message to the caller.

When the caller receives the OK message, it marks the end of the invite transaction. However, the caller is still

required to acknowledge the reception of the OK message by sending an ACK message to the callee. The ACK message is not retransmitted, so if it gets lost, it is the responsibility of the callee to retransmit the OK message.

Note that a stateful proxy becomes responsible for initiating the call once it responds to the caller with the TRYING message in step 2. This lets the caller know that the proxy will handle any retransmissions that are necessary to ultimately reach the callee. Such state maintenance and message retransmission complicates the design of the server, but it decreases the amount of retransmitted messages that the server must process. This potentially allows greater efficiency in handling reliable call establishment, especially when using unreliable protocols, such as UDP, to transport SIP messages. A stateless proxy does not send a TRYING message, because it remains the caller's responsibility to retransmit messages if it does not receive a RINGING message form the callee.

During the INVITE transaction, the caller and callee also negotiate the protocols that will be used for voice data transmission. Once the call is established, voice data transmission occurs directly between the caller and callee (without traversing the proxies) using the negotiated mechanisms.

When the caller or callee wants to hang up, they initiate a bye transaction. The bye transaction consists of the following steps:

1. The caller (callee) sends a BYE message to the proxy indicating that the phone call is complete.
2. The proxy forwards the BYE message to the callee (caller).
3. The callee (caller) acknowledges the BYE message by sending an OK message to the proxy.
4. The proxy forwards the OK message to the caller (callee) and the call is terminated.

By routing the bye transaction through the proxy, the proxy will know when the call has been completed. This is potentially useful for billing or resource management.

3 OpenSER

OpenSER is a widely-used, open-source SIP proxy server. OpenSER receives SIP messages using the network transport protocol, such as UDP or TCP, processes them and finally forwards the SIP message or sends back replies. Processing a SIP message typically involves parsing the message and determining its next destination, possibly involving a database lookup. When OpenSER is configured as a stateful proxy, it stores and maintains all on-going transactions and retransmits SIP messages, as appropriate. In this case, it also needs to access the transaction state while processing every SIP message.

OpenSER must accommodate the particular characteristics of the network transport protocol that is used to deliver SIP messages. In order to understand the effects of the network transport protocol, it is important to discuss the internal architecture of OpenSER. This section describes the architectural similarities and differences when using the different protocols.

3.1 TCP Architecture

Figure 1 shows the high-level architecture of OpenSER when using TCP as the network transport protocol. As the figure shows, OpenSER spawns multiple *worker* processes and a single *supervisor* process. The figure shows the interprocess communication (IPC) between the processes using dotted lines and the access to shared memory objects using dashed lines.

TCP is a connection-based protocol and all communications require that a connection be established first. As this is a relatively expensive operation, it is important to maintain connections across messages within a transaction, or even across transactions. The supervisor process is primarily responsible for managing TCP connections. Each TCP connection has a corresponding application level connection object and these objects are maintained in a shared hash table. As the figure shows, on receiving a new connection request, the supervisor accepts the connection, creates a new TCP connection object, and adds it to the hash table. It, however, does not receive or process any SIP messages. Instead, it assigns the new connection to a worker process, which is then responsible for receiving, processing, and forwarding SIP messages that arrive on that connection. The socket file descriptor of the new TCP connection is passed to the worker process using IPC (dotted lines in Figure 1).

Once a worker process receives a new connection from the supervisor process, it receives all messages that arrive on that connection. As TCP is not message-based, that particular worker is the *only* process that receives messages on that connection. Otherwise, a message might be split across two worker processes. Once a SIP message is received, the worker process processes the message and determines what to do with it. Usually, the worker will determine that it needs to forward the SIP message.

Even though a single worker process is responsible for all SIP messages that arrive over a given connection, transaction state still must be shared among worker processes. This is because TCP connections are delegated to a worker process, not SIP transactions. This means that the connections that compose the two halves of the communication in a transaction are likely to be assigned to different worker processes by the supervisor process. The supervisor can-

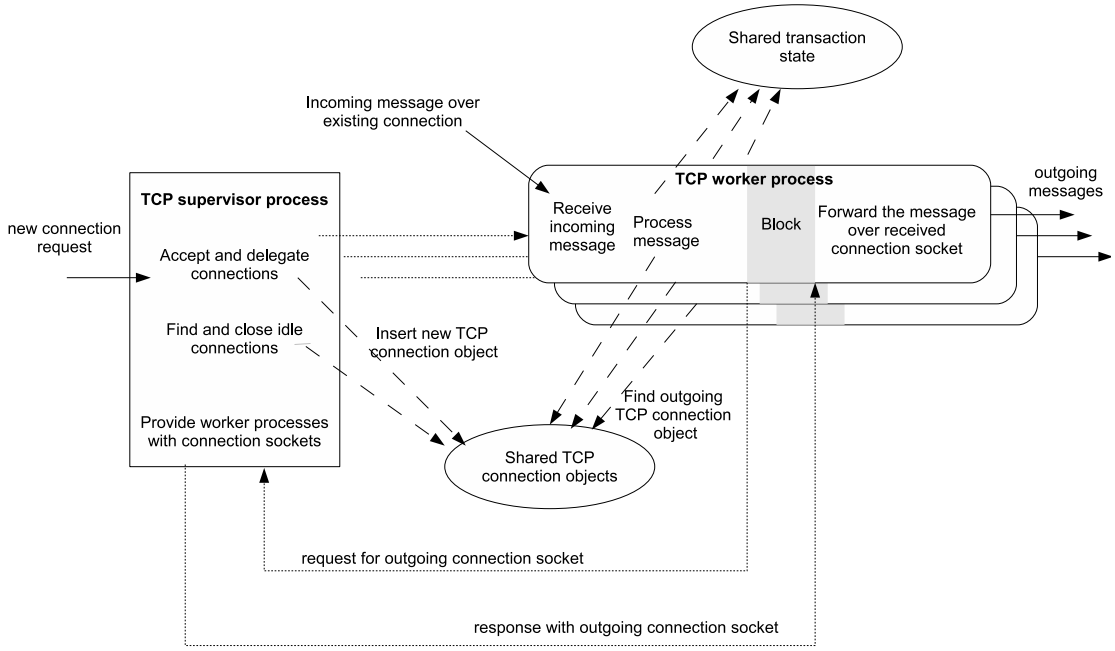

Figure 1. OpenSER TCP Architecture

not know ahead of time which connections will form the two halves of a transaction, and this can change over time. Therefore, there are almost always two worker processes involved in every transaction, one for each direction. This requires synchronized access to the shared transaction state as shown in Figure 1.

In order to forward a message, the worker needs the socket file descriptor of the TCP connection to the receiver. It first obtains the TCP connection object from the shared hash table. Then it requests the socket file descriptor corresponding to this TCP connection object from the supervisor. The supervisor keeps a copy of all of the currently open sockets, regardless of which worker they are assigned to. This allows the supervisor to be able to respond to requests for sockets from the worker processes. As the figure shows, the worker process blocks waiting for a response after it requests a connection from the supervisor process. Eventually, the supervisor responds by passing the socket file descriptor using IPC (dotted lines in Figure 1), allowing the worker process to forward the SIP message over that connection.

The supervisor process is also responsible for closing idle TCP connections. Closing a TCP connection in OpenSER involves two steps, closing all the open socket file descriptors to that connection and destroying the TCP connection object. TCP connections must eventually be closed. If a client chooses not to close the connection immediately after a transaction completes, then the server assumes that the client may want to reuse the connection. So, the server marks the connection as idle. After a certain configurable timeout period, the server closes the connection. Note that at any given time at least two processes, the supervisor and one of the worker processes, have an open socket file descriptor to an active connection. Ultimately, both the worker process and the supervisor must close their socket file descriptors for the connection. To accomplish this, the worker process closes and returns inactive socket file descriptors to the supervisor after a timeout period. The supervisor then waits for an additional timeout period and then closes its socket and destroys the TCP connection object. This requires both the supervisor and the worker processes to check for idle connections.

OpenSER's TCP architecture also includes an additional *timer* process (not shown in the figure). But this process is superfluous for TCP, since TCP itself handles retransmissions.

3.2 UDP Architecture

Figure 2 shows the high-level architecture of OpenSER when using UDP as the network transport protocol. As with the TCP architecture, there are multiple *worker* processes for handling SIP messages. But unlike the TCP architecture, all the processes are symmetric in nature and there does not exist a supervisor process. Each worker process operates independently to receive, process, and forward SIP

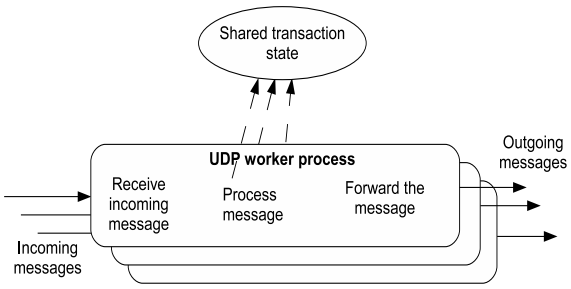

Figure 2. OpenSER UDP Architecture

messages.

Unlike TCP, UDP is a message-based, connectionless protocol. Since UDP is a connectionless protocol, worker processes do not need to establish connections with the clients. Since UDP is a message-based protocol, worker processes do not need to synchronize in order to receive messages. UDP guarantees that when a worker process attempts to receive a message from the network, it will either receive a SIP message in its entirety or nothing at all. Therefore, each worker process can simultaneously receive messages from the network.

In order to process a SIP message, a worker process must first associate the message with a new or ongoing SIP transaction. There is no guarantee, however, that the same worker process will receive all of the messages that make up a single transaction. Therefore, transaction state must be stored in shared memory accessible to all worker processes as shown in Figure 2. Access to this shared memory must be synchronized among the worker processes.

After processing the SIP message, the worker process will typically decide to forward it. With UDP, the worker process can send the complete message out over the network without having to obtain a connection to the client. Similarly, because UDP is message-based, there is no need to synchronize with the other worker processes to send a message. Specifically, even without synchronization, if two worker processes try to simultaneously send a message, to the same or different clients, neither message will interfere with the other.

As with the TCP architecture, the UDP architecture also includes an additional *timer* processes (not shown in the figure). But unlike TCP, the timer process is essential to UDP, since UDP does not guarantee delivery of messages and a stateful proxy must retransmit messages for transactions that do not receive a response. When a message is sent, the worker process creates a new timer which is added to a global list managed by the timer process. The timer process periodically checks the list for timers that have expired. At that point, it accesses the transaction state and retransmits unacknowledged SIP messages. Access to both the global timer list and the shared transaction state must be synchronized with the worker processes.

4 Experimental Setup

4.1 Hardware Setup

The OpenSER VoIP server was evaluated on an AMD-based server system. The server included two 2.4 GHz Opteron 280s (for a total of four cores), 8 GB of memory and four Broadcom NICs. Three of the Broadcom NICs were Gigabit NICs and were used for experiment data, while the other Broadcom NIC was a 100 Megabit NIC, and was used for control purposes. The server was running Ubuntu 7.04 Server Linux 2.6.20-15-generic. OpenSER v1.2 was configured as a stateful SIP proxy using 6500MB of shared memory without the use of TLS. MySQL v5.0.38 was used for persistent storage.

Three client machines were used in the experiments to generate the VoIP calls. Each client machine included two 2.4 GHz Opteron 250s (for a total of 2 cores). Each client machine had a Gigabit network interface, which was connected to the server, and a 100 Mbps network interface for control. All the machines were running Ubuntu 7.04 Server Linux 2.6.20-15-generic. The client machines were monitored closely on every experiment to ensure that they were never the bottleneck.

4.2 Benchmarking Methodology

The benchmark program was designed to simulate thousands of SIP phones across the client machines. A manager program, running on one of the client machines, controlled how many of these phones would make simultaneous calls through the proxy server. The manager also synchronized the callers and the callees independent of the proxy server to ensure that the callees were prepared to receive calls before the callers initiated those calls.

All experiments were conducted in two phases. The first phase consisted of creating all of the SIP phones, which would then each register with the proxy server. This phase is not included in any of the results. Therefore, there are no registration transactions during any of the experiments. Once the manager determines that all phones have successfully registered, it would begin the second phase of the experiment. During this phase, each caller would attempt a set number of calls to its designated callee. Once these calls were completed, both the caller and the callee would report their results to the manager and terminate. This phase is what is being measured in each of the experiments shown in

the results. Proxy throughput is reported as *operations per second*. An operation is a single SIP transaction—either an invite transaction or a bye transaction. There are an equal number of invite and bye transactions in every experiment.

4.3 OpenSER Configuration Issues

There are a few configuration issues that must be dealt with in order to maximize OpenSER's performance. First, the TCP supervisor process must be given an elevated priority level in order to prevent anomalous behavior due to the Linux scheduler. Second, the length of time OpenSER keeps idle connections open must be reduced in order to prevent port starvation under heavy load. Finally, the number of worker processes must be selected to maximize overall performance.

The TCP supervisor process must be given elevated priority because it plays a role in almost every message when using TCP. Whenever a worker must forward a SIP message to either a client or a server, it must first obtain a file descriptor for the connection from the supervisor. In fact the worker process blocks until it gets the response. Thus the supervisor can easily become a bottleneck in the TCP architecture. If the supervisor is not scheduled often enough, there could be multiple stalled worker processes. This in turn could lead to multiple processors being idle, limiting the achievable concurrency, even on a multi-processor system.

Linux 2.6.20 does not automatically schedule the supervisor frequently enough, so this type of starvation occurs regularly. In order to fix this problem, the priority level of the supervisor process was increased to -20, making it the highest priority process. This led to a 40–100% increases in TCP throughput. By elevating the supervisor's priority in this fashion, there is never idle time on the server in any of the experiments, whereas there is idle time if this is not done. Therefore, the priority of the TCP supervisor process is elevated in all of the experiments.

By default, OpenSER keeps idle TCP connections open for 120 seconds. This allows clients to reuse open connections, if they can. In all experiments, the clients never closed their connections, even if they were not going to use them again. This forced the server to keep all connections open for 120 seconds past their last use before closing them. This is a probable usage scenario that maximizes the connection management overhead on the server. However, this caused the server to run out of available ports in many experiments that did not heavily reuse connections. To avoid port starvation, OpenSER was configured to keep idle TCP connections open for only 10 seconds.

Finally, the number of worker processes was selected for the UDP and TCP experiments to maximize overall performance. The server was configured to use 24 worker processes for UDP and 32 worker processes for TCP in all the experiments because it performs well over a wide range of experiments.

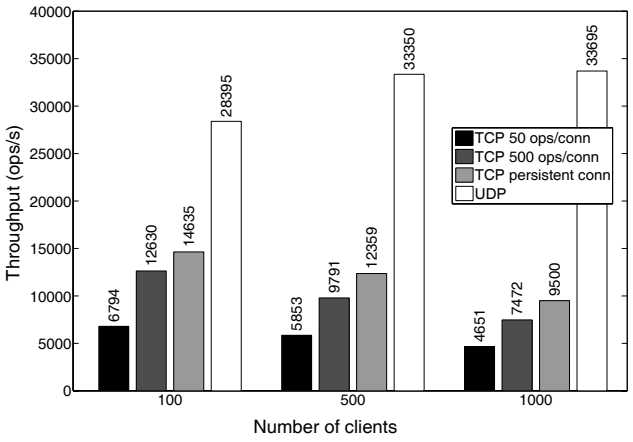

Figure 3. Baseline OpenSER Performance

5 Results

5.1 Baseline OpenSER Performance

Figure 3 shows the difference in OpenSER's baseline performance using UDP versus TCP. Specifically, it shows OpenSER's throughput in terms of SIP transactions per second for each of these protocols. For TCP, three different workloads were used. In the first two workloads the client-server connections are non-persistent and are re-established after 50 and 500 operations. In the third workload, the TCP connections persist for the duration of the experiment, representing the best case scenario from the standpoint of TCP processing and network overheads.

In addition, Figure 3 illustrates how the throughput using each protocol is affected by the number of concurrent clients. Specifically, it presents the different throughputs achieved under load from 100, 500, and 1000 simultaneous clients.

These results clearly show that OpenSER over TCP performs very poorly in comparison to OpenSER over UDP. With 100 clients, the UDP throughput is twice that of TCP under the persistent connection workload. Moreover, these results show that OpenSER's throughput using UDP scales better. As the number of clients increases, the difference increases in UDP's favor. At 1000 clients, there is more than three-fold difference in throughput. These results are consistent with those presented by Nahum *et al.* [10, 11, 12].

The non-persistent TCP connection workloads perform even worse compared to UDP. In the 50 operations per con-

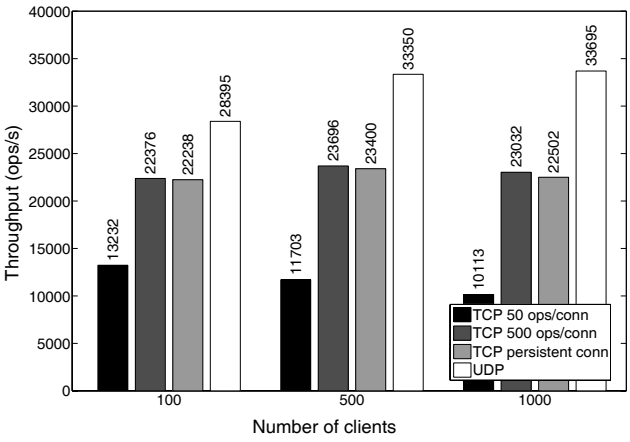

Figure 4. File Descriptor Cache Performance

nection workload, the UDP throughput is about 4 to 7 times the TCP throughput. On increasing the operations per connection to 500, the performance improves, but the UDP performance is still much better with about a four-fold difference in throughput.

The execution profile data from the above experiments, collected using OProfile, showed the IPC between the supervisor and the worker processes to be a major source of overhead. About 12% of the time was spent in the function in which the IPC occurred. Also, a majority of the top fifteen functions in the kernel were IPC related.

This is explained by the fact that, under TCP, immediately after a worker process has forwarded a SIP message, it closes its socket file descriptor for the connection on which it sent the message. Consequently, if the worker processes another message destined for the same client or server, it must again request another socket file descriptor for the same connection from the TCP supervisor process. In fact, this is a common occurrence. For example, it occurs when the same phone both initiates and terminates the same call.

5.2 File Descriptor Caching

Given the potential for the reuse of the socket file descriptors, the impact of a simple file descriptor cache was evaluated. The file descriptor cache is a per-process mapping from TCP connection objects to the socket file descriptors. Before requesting a socket file descriptor from the supervisor process, the worker looks in its cache. If it finds the socket file descriptor, it avoids both the IPC overhead and the time spent waiting on the TCP supervisor process. If it does not find the socket file descriptor, it proceeds with the request to the supervisor and stores the socket file descriptor that it receives in its cache for later reuse.

Figure 4 compares the performance of the TCP implementation with the file descriptor cache to the UDP performance. The file descriptor cache yields a dramatic improvement in the TCP performance and the TCP throughput with the file descriptor cache is within 66-78% of the UDP throughput in the persistent connection case. Further, the TCP implementation with the file descriptor cache scales better than before.

The most significant effect of the file descriptor cache is the reduction in time spent in the function with the IPC from 12.0% to 4.6%. There is also a reduction in time spent in the functions that are used to implement the IPC in the kernel. Notably, after the introduction of the file descriptor cache, the IPC-related functions drop out of the top fifteen functions in the profile, to be replaced by several TCP-related functions. Moreover, the user-level profile with the file descriptor cache looks remarkably like that with UDP.

Figure 4 also shows that in the non-persistent TCP workloads, the results from the 500 operations per connection experiments are very similar to the persistent case. But in the 50 operations per connection case, even though the file descriptor cache helps double the throughput, there is still a two-fold difference in the throughput compared to the other TCP workloads.

The execution profile data from the 50 operations per connection workload experiments showed a surprising reason for the dramatic slowdown with limited duration connections. Specifically, the slowdown does not appear to be a result of connection establishment or tear down overheads inside the kernel. Instead, it appears to be a result of OpenSER's efforts to find and close connections that have not been used in a long time. First, there is almost a three-fold increase in time spent in the function where the supervisor process finds and closes the idle TCP connections. Moreover, throughout the execution of this function, a lock is held on the shared hash table of connections. The effects of this lock can be seen inside the kernel. Specifically, OpenSER uses an implementation of spin locks that calls `sched_yield` after failing to promptly acquire the lock. Notably, the top ten kernel functions are all in the Linux scheduler.

It was also observed that the supervisor process examined every TCP connection object in the shared hash table while holding a lock to identify and close connections that are idle. This was not the only source of overhead, even the worker processes examined every connection they owned. As explained in section 3.1, the worker processes need to return the TCP connections to the supervisor, before the supervisor can destroy the connection.

5.3 Connection Management

A better strategy for finding and closing idle TCP connections would be to keep them sorted based on their time-

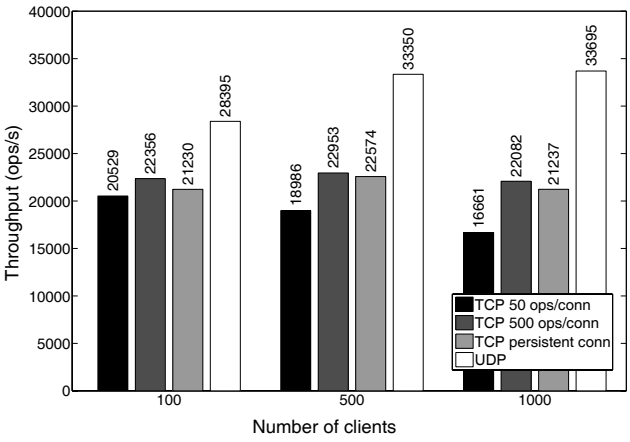

Figure 5. Priority Queue Performance

out values. Once the TCP connections are sorted, the supervisor and the worker processes can easily identify just those connections that have timed out.

Each of the worker processes and the supervisor process use a separate priority queue to manage idle TCP connections. The supervisor process adds all the TCP connection objects in the server to its priority queue. Note that the worker processes also update the timeout value of a TCP connection each time they receive or send a message on that connection. So the workers also require access to the supervisor's priority queue to update the TCP connection object's place in the priority queue. Hence, this priority queue is placed in shared memory. The priority queues used by the worker processes are local to each of them and each contains just the TCP connection objects owned by a worker process.

When checking for idle connections, both the supervisor and the worker processes examine the TCP connection objects from their priority queue until one of them has a timeout period which has not expired. The supervisor cannot destroy a TCP connection object until the connection is returned by the worker process which owns it, hence the supervisor reinserts such TCP connection objects back into its priority queue, even if the timeout period of that connection has expired.

Figure 5 shows the effect of introducing the priority queue. There is a significant impact on the performance in the 50 operations per connection workload, where the throughput is very similar to the other TCP workloads and more importantly it is within 50-72% of the UDP performance. In the other TCP workloads, adding the priority queue has negligible effect on performance since there is very less connection management overhead at the server.

6 Discussion

The results from the previous section show that many difficulties arise in the use of a multi-process architecture for OpenSER. This creates inter-process sharing to deal with transaction state in both the UDP and TCP cases and creates the need to pass connections between processes in the TCP case.

A multi-threaded architecture would solve the latter problem. In a multi-threaded architecture, accesses to transaction state would still need to be synchronized to protect against simultaneous access by multiple threads. This is no different from the multi-process architecture. However, as Figure 4 showed, a lot of the overhead within the TCP architecture arises from transferring connections among processes. File descriptors cannot be shared among processes without passing them back and forth using IPC. This overhead would be completely unnecessary within a multi-threaded server. Locking would still be required to ensure atomic use of each connection, but the threads would be able to use any file descriptor in the server without any expensive transfer operations.

The Stream Control Transmission Protocol (SCTP) [13] has recently emerged as a connection-based alternative to TCP. SCTP allows reliable, message-based communication. Newer versions of OpenSER will support SCTP as a network transport protocol using an architecture similar to the UDP architecture described in Section 3.2. This is possible because SCTP allows connection management to be implemented entirely within the kernel, hiding it from the application. By relieving the application of connection management, several of the overheads found in the TCP architecture of OpenSER would go away. Effectively, the kernel would deal with the use of the same connection by multiple processes and close idle connections efficiently. Additionally, because SCTP is a message-based protocol, user-level locking would not be required to send messages, as sending a message is implicitly an atomic operation.

The TCP architecture of OpenSER exhibits some of the challenges of writing event-driven servers. OpenSER does not correctly use `epoll` (or `select/kevent`) to ensure that it can always read or write sockets. Instead, it sometimes ends up making blocking receive calls, as shown in Figure 1. This can lead to deadlock in the following situation. When a worker process requests a connection from the supervisor process, it then blocks waiting to receive that file descriptor. If, at the same time, the supervisor process blocks waiting to send a new connection to the same worker (since the buffer at the receiver is full), the two processes will deadlock. Once the supervisor process deadlocks, no other worker can make progress either, as they will quickly need their own connections from the supervisor process. Similarly, no new connections will be accepted. This clearly

illustrates that in an event-driven server, one must be careful to only read from sockets when the event mechanism says there is something to read and only write to sockets when the event mechanism says there is space to write.

7 Related Work

Nahum *et al.* extensively analyze the performance of a SIP server, specifically, OpenSER [10, 11, 12]. They study the impact that the choice of the transport layer (UDP versus TCP), the use of authentication, and the stateful or stateless nature of the proxy have on the performance of the server in three SIP scenarios: proxying, registration, and redirection. Specifically, they study the throughput and the latency as a function of load on the server in each of these scenarios under each configuration.

They show that enabling authentication has the most significant impact on performance in all the SIP scenarios and they attribute this to aggressive database lookups. The choice of the transport protocol has the next biggest impact on performance, especially in the proxying scenario. For both UDP and TCP, a large percentage of the calls experience poor response times when the system is overloaded.

They attribute the poor performance with TCP to the complexity of the protocol and the resulting larger code paths. While the TCP performance results are confirmed by this paper, it has been shown that the resulting performance is a consequence of the higher level architecture of the server and not due to the lower level processing overheads.

Cortes, *et al.* [2] examine the main processing elements in a SIP proxy using a number of performance metrics. They also compare different proxy implementations including multi-threaded vs multi-process and single-stage vs multi-stage processing. They show that parsing, string handling, memory allocation, and thread architecture have a significant impact on the performance of the proxy application. However, they use UDP in the transport layer for all their experiments and do not consider the impact of TCP in the proxy server's architecture and the resulting performance.

Janak's thesis [3] describes the design decisions and the implementation details of the SIP Express Router (SER), from which OpenSER is derived. It also explains how the processing of the SIP messages was optimized to improve the server's performance. SIPstone [16], a benchmark for SIP proxy servers, is used to evaluate the performance of the server. It states that the decision to use a multi-process architecture over a multi-threaded or an event-based architecture for SER is primarily based on the ease of programming and portability. It also states that SER was optimized only for UDP at that stage of development.

Others have studied the behavior of the VoIP data stream over the Internet. Markopoulou, *et al.* measure the delay and loss characteristics of the VoIP data stream [8, 9]. They show that some paths through the Internet result in acceptable call quality and some do not. Boutremans, *et al.* also measure the characteristics of the VoIP data stream [1]. They conclude that link failures are the major source of problems, as they lead to long periods of routing instability. Jiang and Schulzrinne examine the effects of bursty packet loss and show the effectiveness of error correcting schemes in recovering the audio data [4]. However, none of these studies consider the behavior or performance of call establishment and completion using the SIP protocol.

Jiang and Schulzrinne have also studied call availability in the Internet [5]. They showed that 0.5% of calls are not established successfully, and an additional 1.5% of calls are aborted. While these experiments do include call establishment and completion, they are targeted at understanding the user experience over the Internet, rather than studying the behavior of VoIP servers.

Koskelainen, *et al.* propose a conference control management system built on top of SIP [7]. SIP is used to provide participant operations between the server and the client. It provides mechanisms to initiate conferences, allow users to join and leave conferences and also ask an outsider to join a conference.

8 Conclusions

The Session Initiation Protocol (SIP) is a widely-used, application-layer signaling protocol for establishing VoIP communication sessions. SIP can utilize a variety of transport protocols, including UDP and TCP. Despite the advantages of TCP, such as reliable delivery and congestion control, the common practice is to use UDP. This practice is founded upon the belief that UDP's lower processor and network overhead will result in much better server performance.

This paper, however, has shown that the principal reasons for OpenSER's poor performance using TCP are, in fact, caused by the server's design, not the low-level overhead differences between UDP and TCP. Specifically, there are two factors that contribute to the significant performance differences exhibited by OpenSER when using TCP compared to UDP. First, the frequency and overhead of interprocess communication for sharing file descriptors among OpenSER's processes has a profound effect on performance. Second, the strategy for managing idle connections is inefficient and time consuming. Once these issues are addressed, OpenSER's performance using TCP is competitive with its performance using UDP. Specifically, we have shown that the TCP performance increases from 13-51% of the UDP performance to 50-78%.

Furthermore, these results strongly motivate the need for

TCP-based SIP proxy servers to be multi-threaded, event-driven servers. This would not have a negative impact on UDP performance. However, we believe that such an architecture would shrink the gap between TCP and UDP performance by streamlining the sharing of connections. With all of the workers in the same address space, they can simply and efficiently share connections, removing the bottleneck of communicating with the supervisor process.

References

[1] C. Boutremans, G. Iannaccone, and C. Diot. Impact of link failures on voip performance. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 63–71, New York, NY, USA, 2002. ACM Press.

[2] M. Cortes, J. R. Ensor, and J. O. Esteban. On SIP Performance. Technical report, Lucent Technologies Inc., May 2004.

[3] J. Janak. SIP Proxy Server Effectiveness. Master's thesis, Czech Technical University, May 2003.

[4] W. Jiang and H. Schulzrinne. Comparison and optimization of packet loss repair methods on voip perceived quality under bursty loss. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 73–81, New York, NY, USA, 2002. ACM Press.

[5] W. Jiang and H. Schulzrinne. Assessment of VoIP service availability in the current Internet. In *Proceedings of the Passive and Active Measurement Workshop*, 2003.

[6] A. Karim. H.323 and associated protocols. Technical report, Nov. 1999.

[7] P. Koskelainen, H. Schulzrinne, and X. Wu. A SIP-based conference control framework. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 53–61, New York, NY, USA, 2002. ACM Press.

[8] A. P. Markopoulou, F. A. Tobagi, and M. J. Karam. Assessment of VoIP quality over Internet backbones. In *INFOCOM 2002: Proceedings of the Joint Conference of the IEEE Computer and Communication Societies*, pages 150–159, 2002.

[9] A. P. Markopoulou, F. A. Tobagi, and M. J. Karam. Assessing the quality of voice communications over Internet backbones. *IEEE/ACM Transactions on Networking*, 11(5):747–760, 2003.

[10] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP Proxy Server Performance. In *NOSSDAV '07: Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, New York, NY, USA, 2007. ACM Press.

[11] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP Server Performance. Technical Report Research report RC24183, IBM T. J. Watson Research Center, Feb. 2007.

[12] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP Server Performance. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 349–350, New York, NY, USA, 2007. ACM Press.

[13] J. Rosenberg, H. Schulzrinne, and G. Camarillo. The stream control transmission protocol (SCTP) as a transport for the session initiation protocol (SIP), RFC 4168. Technical report, Network Working Group, IETF, Jan. 2005.

[14] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol, RFC 3261. Technical report, Network Working Group, IETF, June 2002.

[15] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, RFC 3550. Technical report, Network Working Group, IETF, July 2003.

[16] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone - benchmarking SIP server performance, 2002. http://www.sipstone.org.