One Server Per City: Using TCP for Very Large SIP Servers

Kumiko Ono and Henning Schulzrinne

Dept. of Computer Science
Columbia University, NY 10027, USA,
{kumiko, hgs}@cs.columbia.edu

Abstract. The transport protocol for SIP can be chosen based on the requirements of services and network conditions. How does the choice of TCP affect the scalability and performance compared to UDP? We experimentally analyze the impact of using TCP as a transport protocol for a SIP server. We first investigate scalability of a TCP echo server, then compare performance of a SIP registrar server for two TCP connection lifetimes: transaction and persistent. Our results show that a Linux machine can establish 400,000+ TCP connections and maintaining connections does not affect the transaction response time. This is applicable to other servers with very large TCP connection counts. Additionally, the transaction response times using the two TCP connection lifetimes and UDP show no significant difference at 2,500 registration requests/second in our SIP server implementation. However, sustainable request rate is lower for TCP than for UDP, since using TCP requires more message processing, which causes longer delays at the thread queue for the server implementing a thread-pool model. Finally, we suggest how to reduce the impact of TCP for a scalable SIP server especially under overload control.

Key words. SIP TCP Server Scalability Measurement

1 Introduction

The Session Initiation Protocol (SIP) [1] is used for Internet telephony signaling, i.e., establishing and tearing down sessions. The SIP is a request-response protocol, similar to HTTP, but can work over any transport protocol such as UDP, TCP or SCTP (Stream Control Transmission Protocol) [2]. If SIP messages are sent over connection-less transport protocol, UDP, the SIP server does not have to maintain connection state, and a single socket can be shared to communicate with all the users. UDP seems a better choice to achieve a scalable SIP server in congestion-free networks.

However, TCP is preferred to UDP even in congestion-free networks, since it addresses issues, such as the SIP message size exceeding the MTU (Maximum Transfer Unit), firewall and NAT traversal. Due to its reliable nature, TCP

imposes additional processing cost on the SIP server, i.e., the server has to maintain a TCP socket for each connection. Typically, to facilitate inbound calls to the user phone behind a NAT or firewall, the user phone maintains a persistent TCP connection with the SIP server. It has generally been perceived as difficult for a SIP server to maintain 250,000+ active TCP connections and to keep up with the corresponding number of user registrations and call requests, in order to compete a high-capacity central office, Lucent's 5E-XC$^{\text{TM}}$[3], a high-capacity 5ESS.

Our goal is to measure the impact of TCP on SIP server scalability and performance, and to suggest techniques to maintain a large number of active TCP connections, such as 300,000, on a single server. The remainder of this article is organized as follows. We introduce requirements for SIP servers in Section 3. Then, we show the scalability and performance measurements of an echo server in Section 4 and those of a SIP server in Section 5. We also analyze the reason of the performance differences between TCP and UDP using component tests in Section 6. We conclude with suggestions for reducing the impact of TCP on SIP server in Section 7.

2   Related Work

Since both a SIP server and an HTTP server can use TCP, they face common problems in handling a large number of connections. Kegel [4] aggregates several tips and limits on I/O and event delivery to support more than 10,000 clients for a scalable HTTP server. Libenzi [5] developed the `epoll()` system call and shows that it enables an HTTP server to sustain a high throughput with active 27,000 connections. We built our SIP server on these tips to increase an upper limit of sockets and to enable the server to wait for events on a larger number of connections using the `epoll()` system call. However, we have to consider the differences between a SIP server and an HTTP server as explained in Section 3.

For SIP server scalability, Shemyak and Vehmanen [6] showed that a SIP server can maintain 100,000 inactive TCP connections, emphasizing the effect of using the `epoll()` system call. However, we need to establish the limit for the number of concurrent connections and clarify the bottleneck.

For a scalable SIP server using UDP, Singh and Schulzrinne [7] compared the performance for different software architectures: event-based, thread-pool, and process-pool. They suggested that the process-pool model has the best performance in terms of response time. Additionally, they proposed a two stage architecture, where servers at the first stage dispatch messages to multiple servers at the second stage in order to improve concurrency and reliability. For a highly concurrent server, Welsh et al. [8] proposed a staged event-driven architecture. Each stage contains a thread-pool to drive the stage execution. They showed decoupling load management from service logic increases concurrency with the measurement using 1,024 clients. We discuss the impact of the transport protocol on SIP server scalability, not the impact of the software architecture here.

3 Requirements for a SIP Server

3.1 TCP Connection Lifetime

Although SIP is similar to HTTP, it differs from HTTP in TCP connection lifetime. For example, a SIP proxy server in transaction-stateful mode needs to wait for the response from the User Agent Server (UAS). After ringing, it might take more than 30 seconds for the UAS to answer it. If the server runs in dialog-stateful mode, it needs to wait for the dialog between users to end. Thus, the TCP connection lifetime depends on human response time, and would be much longer than that for HTTP/1.0 [9], but similar to that for HTTP/1.1 [10]. While HTTP/1.0 [9] opens and closes a TCP connection to fetch each embedded object, HTTP/1.1 supports persistent connections across multiple objects by default in order to improve server performance by avoiding unnecessary TCP connection opens and closes, by reducing the impact of TCP slow-start, and by allowing pipelining requests and responses [11]. Typical HTTP clients and servers close inactive TCP connections when the session timeouts. For example, Mozilla Firefox^{TM}sets the session timeout to 300 seconds by default.

However, a SIP UA behind a NAT needs to maintain even an inactive TCP connection in order to wait for incoming calls [12]. In this case, the TCP connection lifetime for SIP would be much longer even than HTTP/1.1. Therefore, the number of sustainable TCP connections and sustainable request rate are crucial factor for the scalability of an outbound SIP server.

3.2 Traffic Model

We assume a target traffic model where a single server accommodates 300,000 subscribers, which is similar scalability to that of Lucent's 5E-XC^{TM}, 256,000 subscribers. Each user quotes their location every 3,600 seconds as defined by default in RFC 3261 [1]. The average call duration is 180 seconds. The traffic is 0.1 erlangs. Thus, the target throughput for registrations is 300,000 BHCA (Busy Hour Call Attempt), which corresponds to 83 requests per second. The target throughput for calls is 600,000 BHCA (= 300,000 * 0.1 * (3,600 / 180)), which corresponds to 167 requests/second. If four mid-call requests, PRACK, ACK, UPDATE and BYE, are also counted as requests, the rate rises to 833 requests/second.

4 Basic TCP Measurements Using an Echo Server

Prior to the measurement for a SIP server, we measured the scalability and performance of an echo server in order to clarify the threshold and bottlenecks in terms of creating and maintaining a large number of concurrent TCP connections. We expected these basic measurements to make it easier to estimate the scalability of a SIP server using TCP.

4.1 Measurement Metrics

First, to establish the limit for the number of concurrent TCP connections on a single server, we measured the number of sustainable TCP connections, memory usage and CPU utilization by the `epoll()` system call. The echo server accepts several TCP connection requests and receives user messages depending on the order of data delivery from multiple echo clients.

Second, we measured the impact of a large number of TCP connections from two perspectives: of establishing and of maintaining active TCP connections. When echo clients send 512 byte messages to the echo server over separate TCP connections. In other words, TCP does not bundle multiple messages into a single packet when sending out.

4.2 Measurement Environment

The server under test (SUT) is an echo server using a single-process and single-thread which runs on a dedicated host with Pentium IV 3 GHz 32-bit dual-core CPU and 4 GB of memory. The SUT runs Linux 2.6.16 configured with the default virtual memory (VM) split, 1G/3G, where the kernel space is 1 GB and the user space 3 GB. We also configured the VM split to 2G/2G for the measurement of the number of sustainable connections only.

For the echo clients, we used ten hosts with Pentium IV 3 GHz 32-bit CPU and 1 GB of memory running Redhat Linux 2.6.9. These hosts communicated over a 100 Mb/s Ethernet connection at light load. The round trip time (RTT) measured by the `ping` command was roughly 0.1 ms.

We configured the SUT and clients to allow a large number of concurrent connections. The upper limit of file descriptors was increased to 1,000,000 at the SUT and to 60,000 at every client. The ephemeral local port range at the clients was expanded to 10,000-65,535, so that each client can establish approximately 55,000 (= 65535 - 1000) concurrent connections.

4.3 Results from Basic TCP Measurement

The Number of Sustainable TCP Connections We measured the number of sustainable TCP connections at three request sending rates, 200, 2,500 and 14,800 requests/second for the echo server. The echo server accepts connection requests from echo clients, exchanges 512 byte messages over each connection, and maintain all the connections. Figure 1 indicates that the echo server can sustain the same number of connections at any request rate of them: approximately 420,000 connections with the default VM split configuration, 1G/3G, and 520,000 connections with the 2G/2G VM split configuration. Figure 1 also shows the overall memory usage and memory usage for TCP socket buffers for the echo server.

The overall memory usage increases linearly at any request rate and the amounts of memory used are approximately 1.0 GB with the 1G/3G VM split configuration, and 1.2 GB with the 2G/2G VM split configuration. However,

memory usage for TCP socket buffers is less than 20 MB at any request rate of them for both VM split configuration. We can deduce that the bottleneck is the amount of memory for TCP connections, which is allocated 2.3 KB per connection as long as the connection remains open, not the amount of socket buffer memory, which is dynamically allocated depending on the request rate.
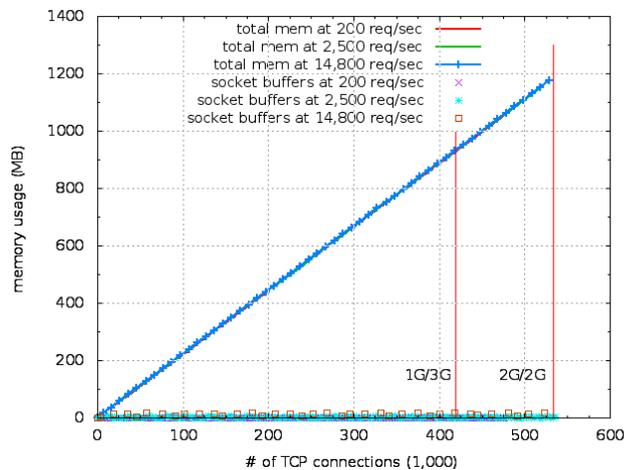

Fig. 1. Memory usage as a function of number of TCP connections at echo server

To get a detailed picture of the memory usage for TCP connections, we monitored the usage of the slab cache, where the Linux kernel configured with a 2G/2G VM split allocates TCP socket data structures including socket buffers at 14,800 requests/second rate.

Figure 2 shows that the slab cache usage for approximately 520,000 TCP connections is 1.2 GB including the data structures for the `epoll()` system call: `eventpoll_epi` and `evenpoll_pwq`. Figure 2 also indicates that the slab cache usage dynamically allocated for the socket buffer heads, i.e., `skbuff_head_cache`, and user data, i.e., `size-512`, is only 12 MB. This result agrees with the result in Figure 1.

Therefore, we have determined that a TCP connection requires 2.3 KB of the slab cache and the bottleneck of sustainable concurrent connections is the amount of allocatable kernel memory for the slab cache, since this slab cache excluding for the socket buffer heads and user data is statically allocated as long as the TCP connection remains open.
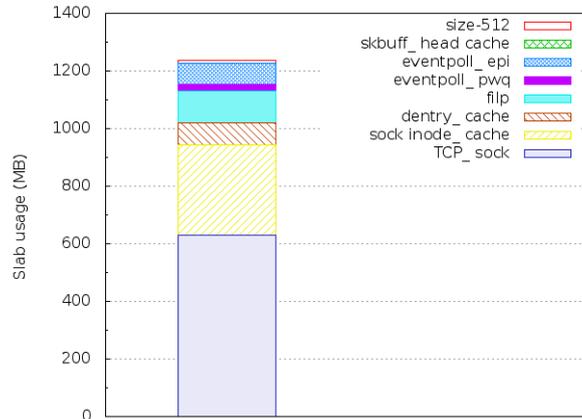
Fig. 2. Slab cache usage for 520,000 TCP connections for echo server

For exchanging TCP control messages and user data, extra amount of the slab cache for socket buffers is required, depending on the request rate and the size of user data. If a target traffic model requires more than 500,000 connections, we recommend to have more than 2 GB of kernel memory, since we have experienced that the server hung without any error message when adding concurrent TCP connections because of memory exhaustion.

In later versions of Linux, e.g., Linux 2.6.20, the system produces an error, "out of memory", when it tries to allocates kernel memory for the TCP socket data structures.

To increase kernel memory, installing more physical memory for a 32-bit kernel does not help since the kernel process can only handle 4 GB of memory including user space. The only way to increase kernel space for a 32-bit kernel is to modify the memory split to 3G/1G, where kernel space is 3 GB. Another way is to switch to a 64-bit kernel. Once kernel can support more than 4 GB of physical memory for a 64-bit kernel, the bottleneck would move to other factors, such as the number of file descriptors, which is currently 1024*1024.

The Cost of Establishing TCP Connections Figure 3 compares the response time and peak CPU time across different connection lifetimes for TCP and to UDP at two request rates, 2,500 and 14,800 requests/second. Transaction-based TCP opens a TCP connection before sending a user data, i.e., a 512 byte

message in ANSI text, and closing the TCP connection after receiving the echoed message. Persistent TCP has two scenarios: with open, where the echo server opens and maintains TCP connections, and without open, where the echo server reuses existing connections.

Comparing the results between the two persistent TCP scenarios indicates the cost of establishing a new TCP connection, which contains a RTT. This costs 0.2 ms of the response time in our local network. Since the RTT delay strongly depends on the network condition, the difference of the response times would be larger in a wide area network. Establishing a new TCP connection also costs 15 percent of CPU time only at high request rate, not at low request rate. We deduce that this difference is caused by the relationship between the required processing rate and the capacity of the CPU cycle. The SUT can process the requests at 2,500 requests/second rate, under the capacity of the CPU, but not at 14,800 requests/second rate. However, even at high request rate, these CPU cost is not so significant, since the maximum CPU time of our server running on a dual-core CPU is 200 percent.

Comparing the results between transaction-based and persistent TCP with open indicates the cost of closing a TCP connection. This costs a negligible amount of the response time and 14 percent of CPU time at high request rate.

Thus, the cost of establishing TCP connections is not significant at low request rate, 2,500 requests/second, which is significantly above the requirement. Furthermore, up to a request rate of 14,800 requests/second, the amount of kernel memory, rather than CPU cycles, limits the scalability of the echo server, as we have determined in Section 4.3.

The Cost of Maintaining TCP Connections Figure 4 shows the response time consisting of the TCP handshake and message exchange as a function of the number of concurrent TCP connections for the echo server. The echo server establishes new TCP connections at 14,800 connections/second and leaves them. The "handshake" data points show the elapsed time for the TCP three-way handshake to establish a new connection, and the "send-recv" data points show the interval between sending and receiving an echoed message after the handshake. The "total" data points shows the sum of them.

Regardless of the number of maintaining TCP connections, the response time remains constant around at 0.3 ms for 'send-recv", which matches the results of persistent TCP in Figure 3, and at 0.4-0.5 ms for "total", of which average matches the results of persistent TCP with open. Thus, maintaining TCP connections affects neither the performance of establishing new TCP connections nor of exchanging user data.

5 SIP Server Measurements

From the results of the basic TCP measurements, we have determined that TCP impacts mainly on kernel memory. Although each TCP connection consumes 2.3 KB of kernel memory, establishing and maintaining 300,000 TCP connections
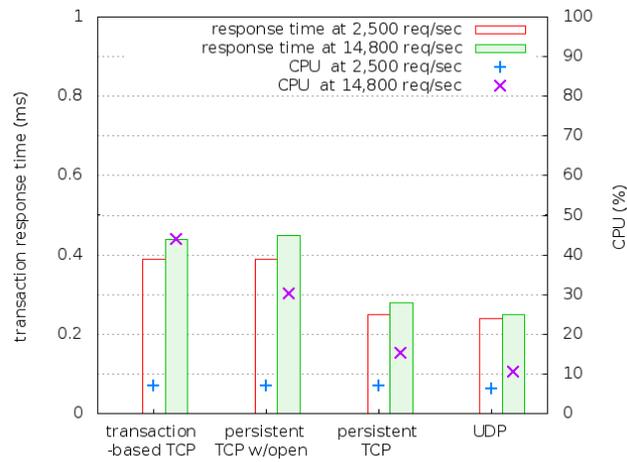
Fig. 3. Transaction response times in left axis and CPU utilization in right axis for echo server
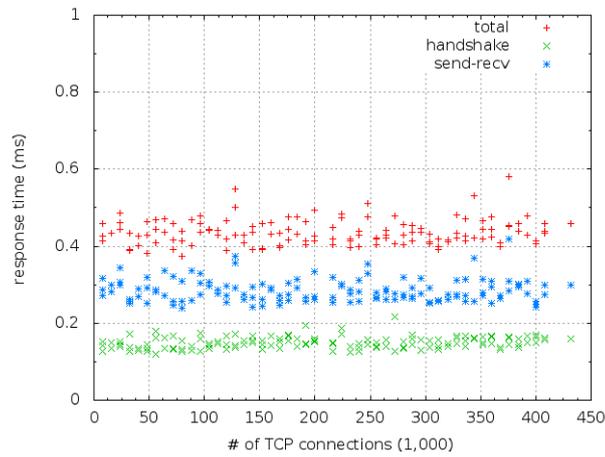

Fig. 4. Response times as a function of number of TCP connections for echo server

themselves does not significantly affect the performance. Compared to the echo server, a SIP server requires no additional kernel memory. Thus, the TCP impact on kernel memory is the same for a SIP server. Therefore, we can focus on measuring the performance for a SIP server.

Although RFC 3261[1] does not strictly define that a SIP server and UAs support persistent TCP connections, we can assume that a SIP server supports persistent TCP connections, but SIP UA behaviors may vary. Thus, we need to clarify how TCP connection handling affects throughput on a SIP server and data transfer latency, i.e., the sustainable request rate and the transaction response time. We measured them for two cases of TCP connection lifetime: transaction and persistent. These cases of TCP connection lifetime differ in how many SIP messages share a connection and how often TCP connections are established and closed.

Transaction-based TCP: UAs create new TCP connections for each transaction, e.g., REGISTER-200 OK, BYE-200 OK, INVITE-200 OK, ACK, BYE-200 OK, UPDATE-200 OK. For the average call or dialog, four TCP connections are established and closed. The maximum transaction duration with the default configuration in [1] is 32 seconds.

Persistent TCP: UAs and SIP servers keep TCP connections created when sending REGISTER requests, and reuse them to send INVITE requests or to update the registration. The default registration interval is 3,600 seconds.

5.1 Measurement Environment

The SUT is our SIP server, sipd [13], running on the same host used for the echo server in Section 4.2. The sipd SIP server implements a single process and a thread-pool model, where a fixed number of threads is spawned on startup, is pooled, and handles tasks upon requests. If more tasks are requested than the number of threads, the tasks wait in a queue. The SIP server has the registrar and proxy functions. User information including registered locations is stored in a MySQL DBMS running on a different server on the same local network. For SIP UAs, we used a SIP UA emulator, part of sipstone test suite [14], running on the same hosts used for the echo clients in Section 4.2. Figure 5 shows these entities and message exchanges. We applied a scenario for a registration test, i.e., REGISTER-200 OK test, not a call test, i.e., INVITE-200 OK test, here. The registration test is similar to the basic measurement using echo server except parsing messages and SIP operation, while the call test is more complicated especially in terms of the number of SIP messages and transactions.

5.2 Registration Test Scenario: REGISTER-200 OK Test

For the registration test, 30,000 UACs send REGISTER requests at various request rate and receive the 200 OK response from the SIP server.

We measured the transaction response time of REGISTER-200 OK for two connection lifetimes: transaction in Figure 6 and persistent in Figure 7. Under
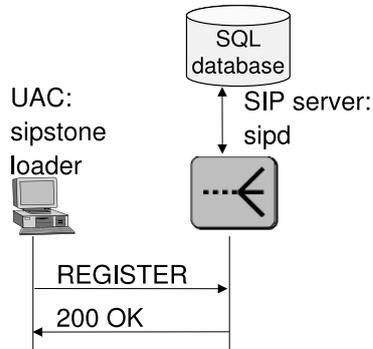
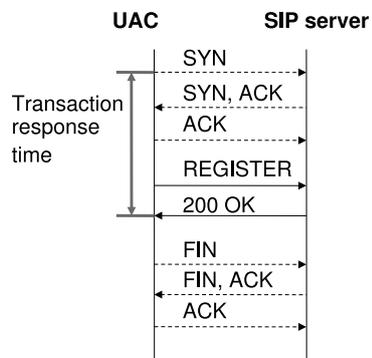**Fig. 5.** Message exchanges for SIP server measurement

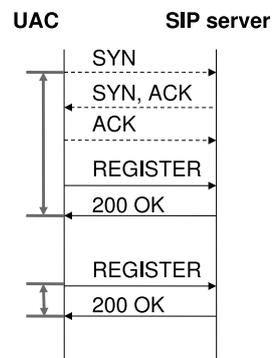**Fig. 6.** Transaction-based TCP message exchanges for REGISTER-200 OK test

Fig. 7. Persistent TCP message exchanges for REGISTER-200 OK test

persistent TCP, we measured the response time in two cases: initial registration that requires TCP connection establishment, which is corresponding to persistent TCP with open for the basic TCP measurement, and subsequent registration that reuses the existing TCP connection.

5.3   Results from REGISTER-200 OK Test

Figure 8 compares the transaction response times at various request sending rates at 100% success rate. The sustainable request rate for transaction-based TCP is 2,900 requests/second, that for persistent TCP with open is 3,300 requests/second, that for persistent TCP is 4,100 requests/second, and that for UDP is 5,300 requests/second. Below 1,600 requests/second, the gaps in their response times remain constant, but above that, the gaps enlarge exponentially. This exponential increase of the response time conforms to Little's theorem. Since sipd has a M/D/c queue for handling tasks, we can deduce the increase of the response time is caused by waiting tasks' exceeding the number of pooled threads (= c) at these request rates.

To investigate these gaps more closely, we compare the response time and CPU utilization at 2,500 requests/second sending rate with those of the basic TCP measurements in Figure 3, as shown in Figure 9. Since the number of messages and transactions are same, this comparison indicates the cost of handling SIP requests: message parsing and SIP operations or the difference of the software model. The cost of handling SIP requests in CPU time is 15-18 percent for all cases, and the cost in the transaction response time is 0.4-1.2 ms. These cost gaps among three TCP cases and UDP increase more in the transaction response time than those in CPU time. For example, the difference in the transaction response time between the two persistent TCP cases, indicating the cost of establishing a TCP connections, is 0.4 ms, which is 0.2 ms in the basic TCP measurement. Thus, we determined that this increased cost of establishing TCP connections were caused by the software model of the SIP server. The bottleneck of sustainable request rate is the thread queue, where the number and lifetime of threads cause queuing delay of threads in the thread-pool model. Section 6 shows the result of component tests that focus on threads in sipd to investigate this reason.

Figures 10 and 11 show that the SIP server starts to fail in handling SIP requests far before exhausting system resources for persistent TCP and UDP. Although we omit presenting the results of transaction-based TCP and persistent TCP with open, their results are similar to that for persistent TCP except the sustainable rate. For all cases, CPU utilization is still below 40 percent and usage of physical memory in RSS and virtual memory in VSZ is below 200 MB and below 800 MB, respectively. Clearly, the bottleneck is neither memory usage nor CPU utilization.

When the success rate drops, the SIP server produces warning messages saying that the overload control drops 83% of requests for persistent TCP and 10-28% of requests for UDP. Since sipd detects overload by monitoring the thread
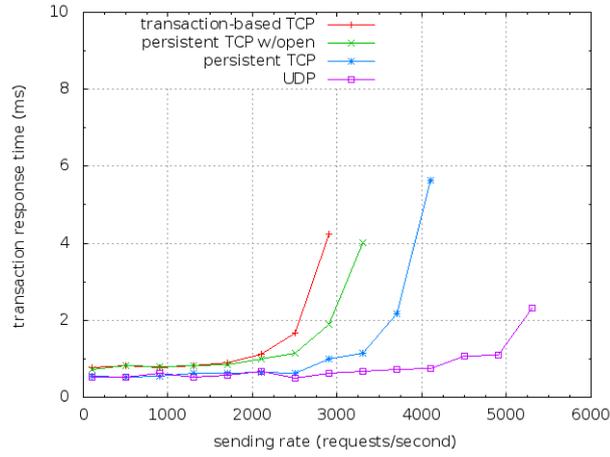
Fig. 8. Transaction response times as a function of sending rate for REGISTER-200 OK test for TCP and UDP
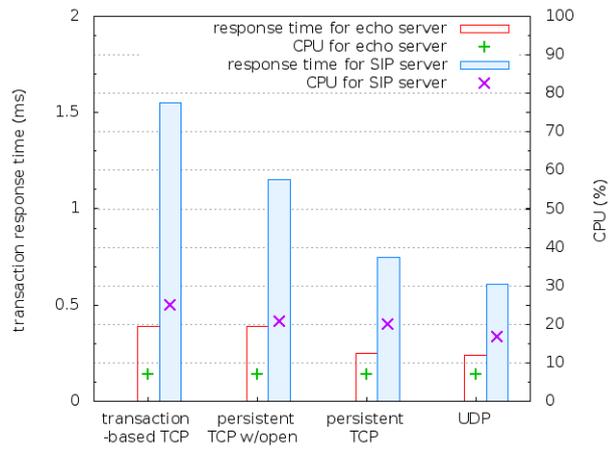

Fig. 9. Transaction response times in left axis and CPU utilization in right axis for REGISTER-200 OK test at 2,500 requests/second
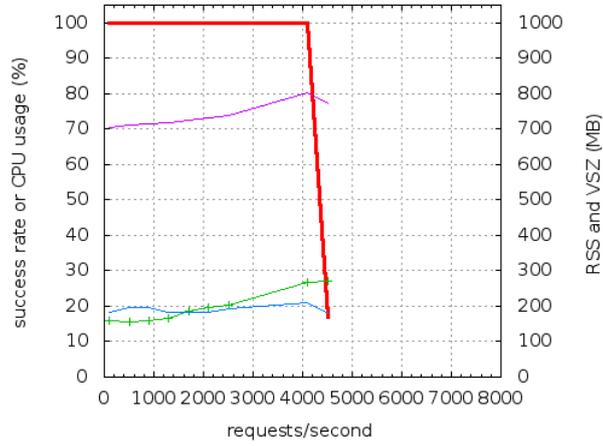
Fig. 10. Success rate, CPU utilization and memory usage for REGISTER-200 OK test: persistent TCP
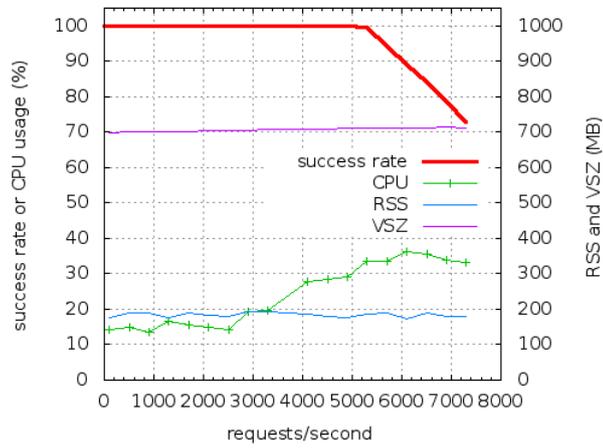

Fig. 11. Success rate, CPU utilization and memory usage for REGISTER-200 OK test: UDP

queue of waiting tasks for available threads, we have determined the bottleneck is the thread queue.

Furthermore, the success rate for persistent TCP steeply drops, while that for UDP gradually decreases. The difference between persistent TCP and UDP in droping the success rate implies that the overload control works ruthlessly for TCP, while it works gracefully for UDP. The overload control at sipd sets threshold of tasks in the thread queue and drops tasks to handle new requests excluding BYE requests, which have fewer subsequent SIP messages. To investigate the difference between TCP and UDP, Section 6 discusses the details of the overload control mechanism.

6 Component Tests

From the results of the SIP measurement, we have determined that the major cause of the difference in the sustainable request rate between TCP and UDP is message processing in a thread-pool model, rather than socket handling. Also, we have determined the overload control for the SIP server works worse for TCP than for UDP.

Since TCP is a connection-oriented protocol, the SIP server needs to handle control messages for the connections, such as TCP open and close requests. This requires more messages to be handled. Also, since TCP transfers byte-stream data, the SIP server needs to find the end of the message by parsing. This requires longer thread lifetime that causes lower throughput. Furthermore, this makes overload control harder, since it disables sorting messages by parsing the first line of the message before parsing the whole message. To confirm these analysis, we performed component tests to focus on message processing.

6.1 Message Processing Test

We performed the REGISTER-200 OK tests as a white-box test, i.e., measuring the called times and the elapsed time of the functions involved in message processing. To avoid the influence of queuing, we set the load low to 10 requests/second, and ran the test for 10 seconds.

Figure 12 compares the number of function calls and new threads required for sipd to process a REGISTER message. The base thread, not a new thread, processes sockets, e.g., calling the `accept()` system call to create a new connection. For TCP, a new thread reads buffer and parses a message. Transaction-based TCP requires most function calls for processing sockets, and most threads for reading buffers, since it receives TCP-SYN and FIN. Although receiving TCP-SYN does not require to read buffer, FIN require to read a zero-sized buffer. Persistent TCP with open requires the second most function calls since it receives TCP-SYN. For UDP, on the other hand, the base thread reads buffer and parses the first line to sort messages for the overload control, then a new thread parses a message again for SIP operations. This makes the overall elapsed time

for UDP slightly longer than that for persistent TCP as seen in Figure 13, although persistent TCP and UDP require the same number of function calls and threads.

However, the elapsed time for reading and sorting messages for the overload control for UDP is one forth of that for persistent TCP, since sorting message for UDP limits the number of lines to be parsed to one. Furthermore, the elapse time for parsing message by a new thread is slightly shorter for UDP than for persistent TCP, since reading buffer has already been processed by the base thread for UDP. Therefore, we can determine that these two differences cause the better sustainable rate for UDP than for persistent TCP in the registration test, although these differences in the thread lifetime is much smaller that the elapsed time of SIP operation, which dominate in the elapsed time. The cost of sorting message for the overload control makes the SIP server performance significantly worse for TCP at high loads.
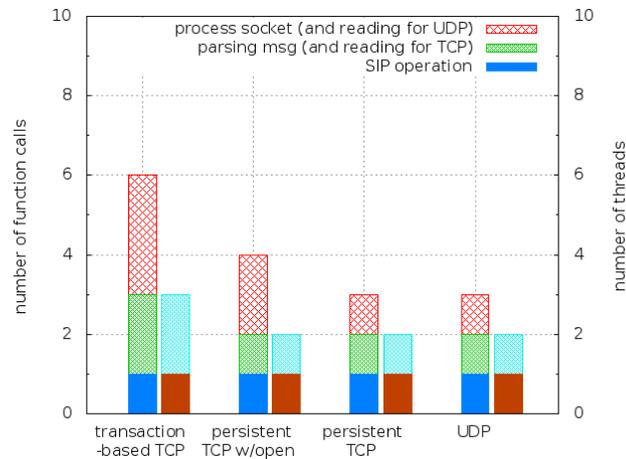
Fig. 12. Number of function calls in left axis and threads in right axis for reading and parsing a REGISTER message
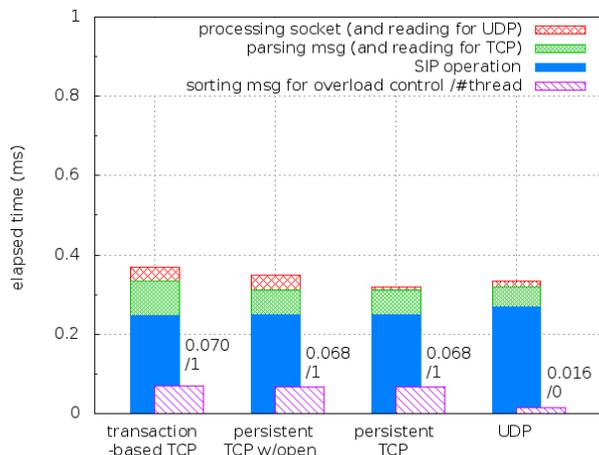
Fig. 13. Elapsed times for reading and parsing a REGISTER message

7 Suggestions for Reducing the Impact of TCP on a SIP Server

Under our target traffic model, we can conclude that the impact of TCP on the scalability of a SIP server is relatively small, since it only includes the setup delay for the TCP three-way handshake and 690 MB of kernel memory for 300,000 concurrent TCP connections. However, as HTTP/1.1 defines persistent TCP to improve the HTTP server performance, persistent TCP is also recommended to avoid unnecessary the setup delay for a SIP server.

Under heavy loads, however, persistent TCP is not efficient enough to compete with the sustainable rate for UDP, since a SIP server falls to overload condition earlier than using UDP. We suggest some approaches to reduce the impact under heavy load.

7.1 Accelerating Parsing for Overload Control

We first suggest that the SIP server sort messages by parsing the first-line of a buffered message without determining the exact message boundary. As found in Section 6, the sorting message for UDP, which is by parsing the first line of the message is much lighter than that for TCP. The speeding up of sorting messages can make it easy to process the overload control for the SIP server.

Although this sorting is accurate not for all messages, it works mostly. The messages with a higher priority for the overload control, i.e., responses and BYE request, are relatively short in size. Thus, the message is unlikely to be sent partially. As a result, the size of receiving buffer to be read by user applications is usually large enough to buffer a SIP message at once. Thus, with a high possibility, the SIP server can parse the first line without determining the message size by parsing the Content-Length header. Even if the SIP server cannot determine the message type because of partial delivery or bundled delivery, the server can simply drop such a message fragment under overload.

Another suggestion is that a function required for the overload control, such as sorting messages, be processed by the base thread that does not need to wait for an additional thread. This is only applicable in a thread-pool model.

Furthermore, the software architecture should handle many concurrent requests efficiently. Rather than the thread-pool model like our SIP server, a small set of multiple processes running a single thread each is more appropriate to avoid causing large queuing delay and unnecessary context-switching.

8 Conclusion

We have shown measurement results to clarify the impact of TCP on SIP server scalability and performance. Choosing TCP requires 2.3 KB of kernel memory per TCP connection and additional CPU cycles mainly for the TCP handshake. Establishing TCP connections causes a setup delay of 0.2 ms in our environment, while maintaining TCP connections only consumes kernel memory. Thus, maintaining 300,000 TCP connections requires approximately 690 MB. These results are applicable to other TCP servers.

The impact on the response time for our SIP server is not significant under our target traffic model. However, under heavy loads, e.g., 2,900 requests/second for the registration test, the major impact is on the transaction response time and on the success rate. The response time exponentially increases around the sustainable rate in our SIP server implementation. This increase is caused by queuing delay in the thread pool model, when thread queues exceeds the maximum length. To avoid this, the software architecture should be selected to achieve a large number of concurrent requests. Above the sustainable rate, the success rate drops steeply by the overload control for our SIP server. From the results of the component tests, we suggest to speed up message parsing to ease overload control for a SIP server. By easing overload control, the SIP server could sustain a much higher request throughput beyond our target traffic model.

References

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

2. R. Stewart. Stream Control Transmission Protocol. RFC 4960, IETF, September 2007.

3. Lucent-Alcatel. Lucent Technologies new high-capacity switch accelerates cost-effective migration to Internet Protocol networks (news release). `http://www.alcatel-lucent.com/`, December 2002.

4. D. Kegel. The C10K problem. `http://www.kegel.com/c10k.html` (accessed in January 2006).

5. D. Libenzi. Improving (network) I/O performance. `http://www.xmailserver.org/linux-patch es/nio-improve.html` (accessed in January, 2006).

6. K. Shemyak and K. Vehmanen. Scalability of TCP Servers, Handling Persistent Connections. In *Sixth International Conference on Networking (ICN'07)*, April 2007.

7. K. Singh and H. Schulzrinne. Failover and Load Sharing in SIP Telephony. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2005.

8. M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

9. T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, IETF, May 1996.

10. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

11. H.F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *ACM SIGCOMM '97*, September 1997.

12. C. Jennings and R. Mahy. Managing Client Initiated Connections in the Session Initiation Protocol(SIP). Internet-draft, IETF, November 2007. `http://www.ietf.org/internet-drafts/draft-ietf-sip-outbound-11.txt`.

13. J. Lennox, H. Schulzrinne, and et al. Cinema:sipd. `http://www.cs.columbia.edu/irt/cinema/doc/sopd.html`.

14. S. Narayanan, A Yu, T Kapoor, and H. Schulzrinne. Sipstone test suite. `http://www.cs.columbia.edu/IRT/cinema/sipstone`.