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

Kumiko Ono and Henning Schulzrinne
Dept. of Computer Science, Columbia University.
Email: {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 server for three TCP connection lifetimes: transaction, dialog, and persistent. Our results show that a Linux machine can establish 450,000+ TCP connections and maintaining connections does not affect the transaction response time. Additionally, the transaction response times using the three TCP connection lifetimes and UDP show no significant difference at 2,500 registration requests/second and at 500 call requests/second. However, sustainable request rate is lower for TCP than for UDP, since using TCP requires more message processing. More message processing 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. This is applicable to other servers with very large connection counts.

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. Appendices describe system calls, system configurations and measurement tools.

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 achieve 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 one 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 describe the impact of the transport protocol on SIP server scalability, not the impact of the software architecture.

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 sets the session timeout to 300 seconds by default.

However, a SIP client 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 [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. Figure 1 shows that the echo server with polling accepts several TCP connection requests and receiving user messages depending on the order of data delivery.

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. Appendix C describes the measurement tools for these metrics.

4.2 Measurement Environment

Figure 2 illustrates our measurement environment consisting of the server under test (SUT) and the echo clients. The 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 a 2G/2G virtual memory

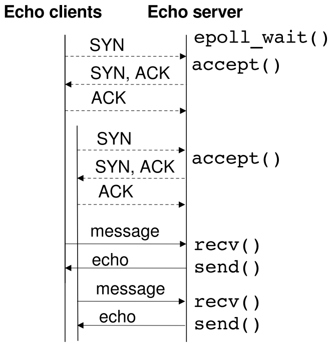

Figure 1: Message exchanges for echo server measurement

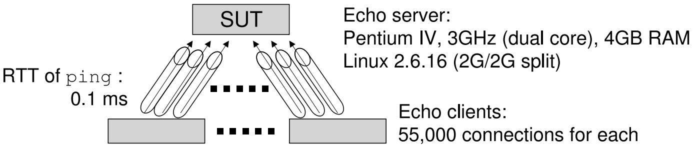
SUT
Echo server:
Pentium IV, 3GHz (dual core), 4GB RAM
Linux 2.6.16 (2G/2G split)
RTT of ping :
0.1 ms
Echo clients:
55,000 connections for each

Figure 2: TCP measurement environment

split, where the kernel can use 2 GB of memory. 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 `ping` 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. Appendix B details the system configurations.

4.3 Results from Basic TCP Measurement

4.3.1 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 in the message sequence shown in Figure 1. Figure 3 indicates that the echo server can sustain approximately 520,000 connections at any request rate of them. Figure 3 also shows the overall memory usage and memory usage for TCP socket buffers for the echo server. The "high" line indicates the system limit of TCP socket buffer memory, 800 MB, which is automatically configured at boot time based on available memory. The overall memory usage increases linearly at any request rate and the amount of used memory is approximately 1.2 GB. However, memory usage for TCP socket buffers is less than 20 MB at any request rate of them. 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.

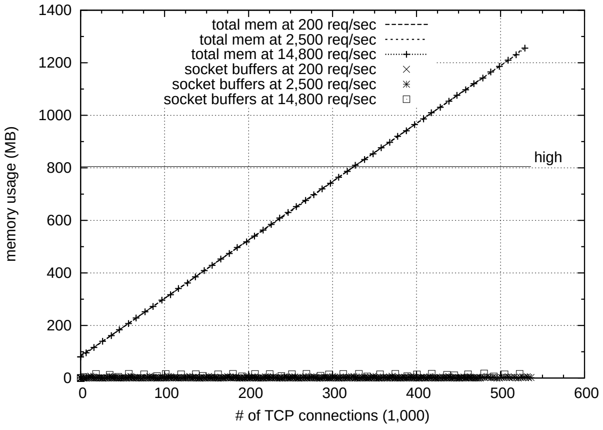

Figure 3: Memory usage as a function of number of TCP connections at echo server

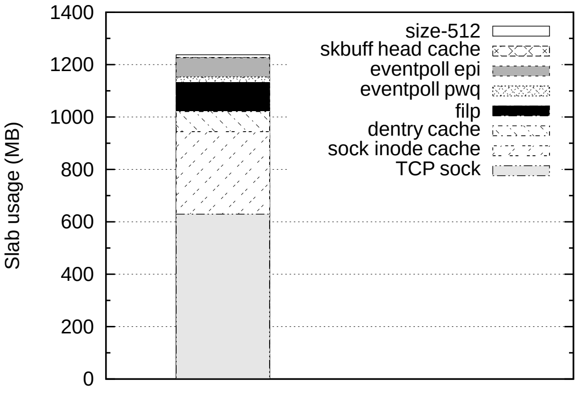

Figure 4: Slab cache usage for TCP connections for echo server

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

Figure 4 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 4 also indicates that the slab cache usage dynamically allocated for the socket buffer heads and user data is only 12 MB. This result is coherent with the result in Figure 3.

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 is statically allocated as long as the TCP connection remains open. 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”, then the “oomkiller” process kills heavy processes including the echo server process 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, the bottleneck would move to other factors, such as the number of file descriptors, which is currently 1024*1024.

4.3.2 The Cost of Establishing TCP Connections

Figure 5 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. This costs 0.2 ms of the response time and 15 percent of CPU time at high request rate in our measurement environment. 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. Since the maximum CPU time of our server running on a dual-core CPU is 200 percent, these CPU cost is not so significant.

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 4.3.1.

4.3.3 The Cost of Maintaining TCP Connections

Figure 6 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 5, 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 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.

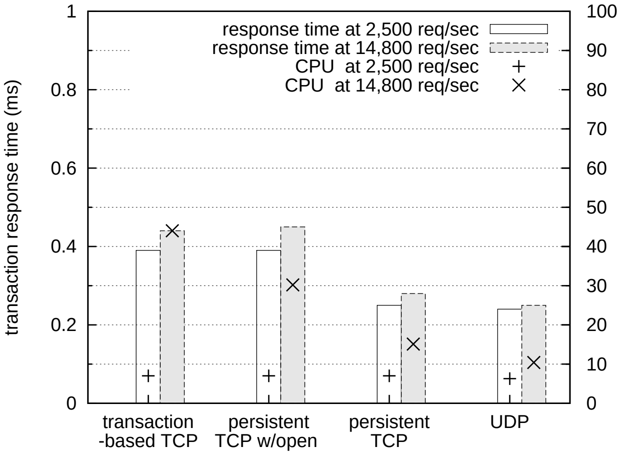

Figure 5: Response times and CPU utilization for echo server

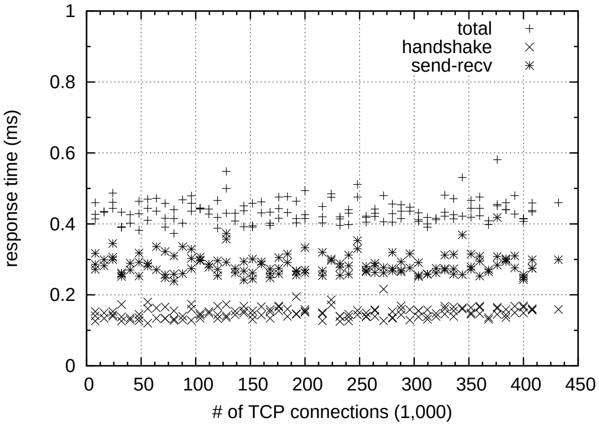

Figure 6: Response times as a function of number of TCP connections for echo server

Although [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 latency, i.e., the sustainable request rate and the transaction response time. We measured them for three cases of TCP connection lifetime: transaction, dialog and persistent. These three 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.

Dialog-based TCP: UAs create and share new TCP connections for a dialog, i.e., from sending INVITE requests to exchanging BYE requests and the 200 OK responses. In our traffic model, a TCP connection is maintained for 180 seconds on average.

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 as 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 proxy and registrar functions are co-located on a single server to measure the effect of reusing TCP connections established at registration. The SIP server runs as an outbound proxy for both a User Agent Client (UAC) and the User Agent Server (UAS), and also as their inbound proxy for simplification. In other words, we used a single server model, instead of trapezoid model, where two servers connect to UAs and interconnect with each other. 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 as the echo clients in Section 4.2. Figure 7 shows these entities and message exchanges. All SIP requests and responses traverse the SIP server to emphasize the impact of the TCP connection for the SIP server. By dividing the sequences into two, we applied two scenarios of tests: a registration test, i.e., REGISTER-200 OK test, and a call test, i.e., INVITE-200 OK test. Table 1 compares these two tests to the basic TCP measurement using echo server. 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.

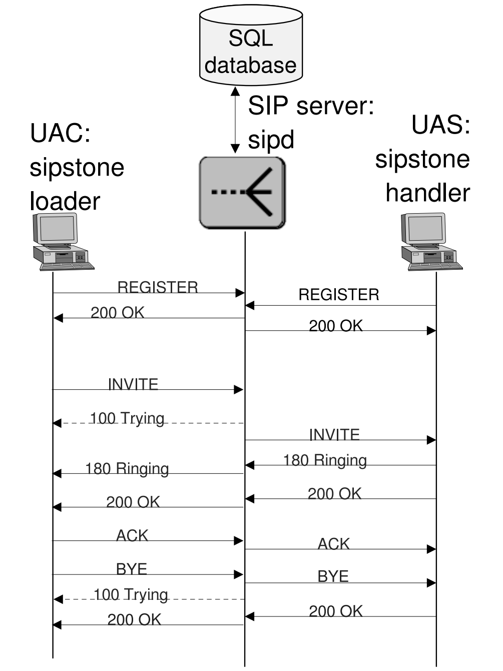

Figure 7: Message exchanges for SIP server measurement

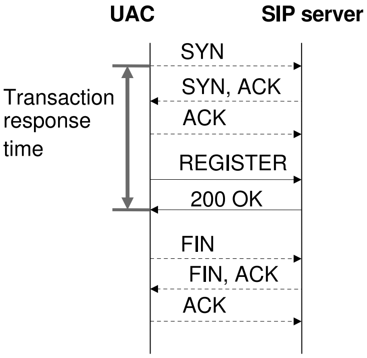

Figure 8: Transaction-based TCP sequence for REGISTER-200 OK test

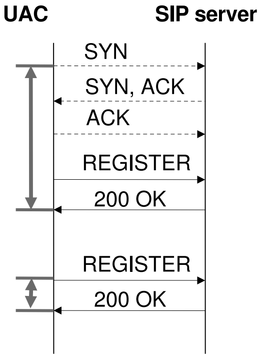

Figure 9: Persistent TCP sequence for REGISTER-200 OK test

5.2 Registration Test Scenario: REGISTER-200 OK Test

For the registration test, the UAC sends REGISTER requests and receives the 200 OK response from the SIP server. Since registrations create no dialog, we measured the transaction response time of REGISTER-200 OK for two connection lifetimes: transaction in Figure 8 and persistent in Figure 9. Under 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 10 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/1 queue for

Table 1: Comparison between TCP measurements for echo server and SIP server

		Echo server	SIP server	
			Registration test	Call test
Processing at server	processing sockets	Yes	Yes	Yes, including sockets to UAS
	reading buffer	Yes	Yes	Yes
	parsing messages	No	Yes	Yes
	SIP operation	No	Yes, including DB access	Yes, including DB access
Number of SIP messages per request		2 (rcv 1, snd 1)	2 (rcv 1, snd 1)	14 (outbound: rcv 3, snd 5, inbound rcv 3, snd 3)
Number of transactions per request		1	1	6 (outbound 3, inbound 3)
Software model		Single process/thread	Single process, thread-pool	Single process, thread-pool

handling tasks, we can deduce the increase of the response time is caused by waiting tasks' exceeding the maximum queue length 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 5, as shown in Figure 11. Since the number of messages and transactions are same as shown in Table 1, 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 12 and 13 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.

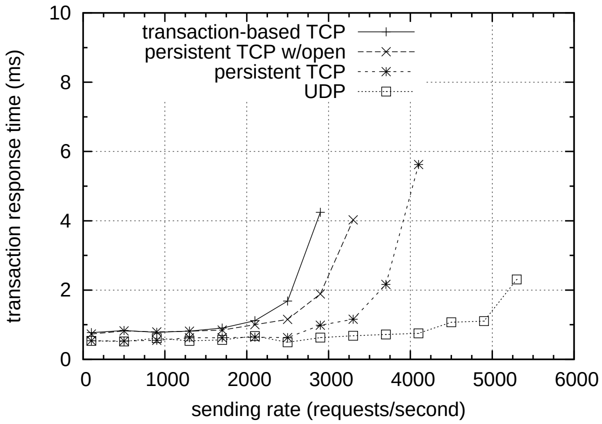

Figure 10: Response times as a function of sending rate for REGISTER-200 OK test for TCP and UDP

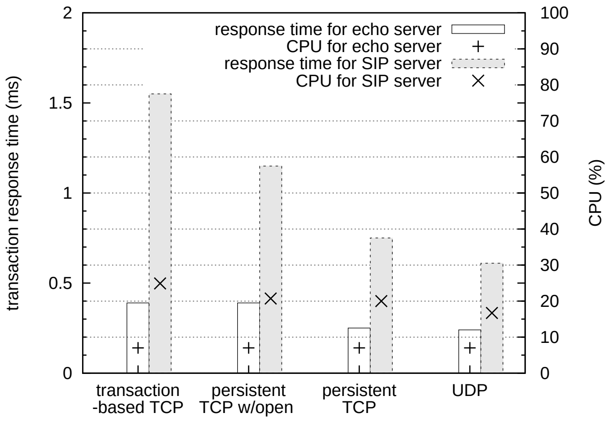

Figure 11: Response times and CPU utilization for REGISTER-200 OK test at 2,500 requests/second

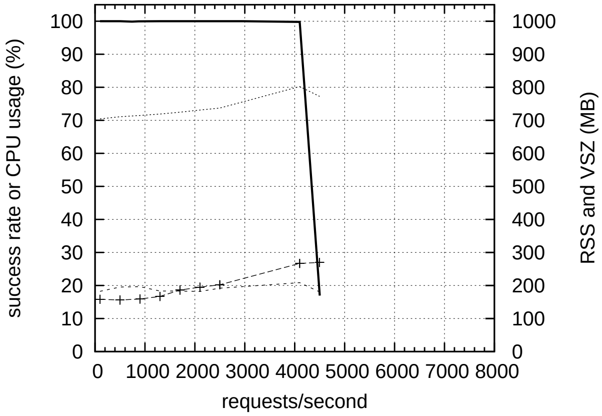

Figure 12: Success rate, CPU utilization and memory usage for REGISTER-200 OK test: persistent TCP

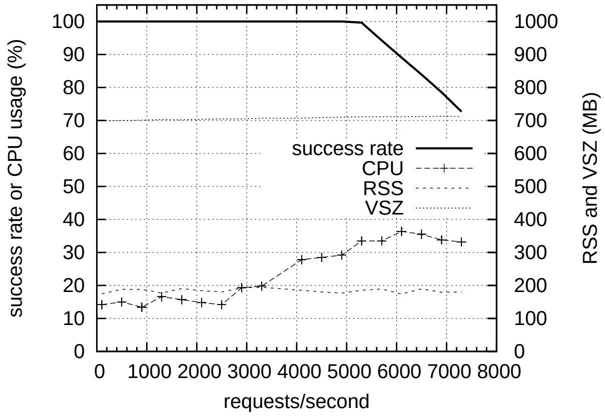

Figure 13: Success rate, CPU utilization and memory usage for REGISTER-200 OK test: UDP

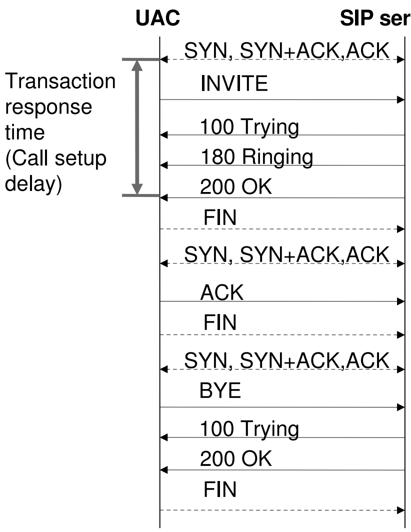

Figure 14: Transaction-based TCP sequence for INVITE-200 OK test

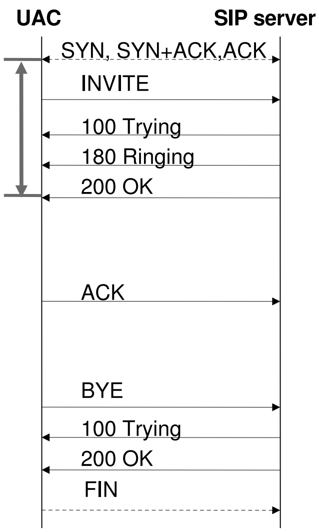

Figure 15: Dialog-based TCP sequence for INVITE-200 OK test

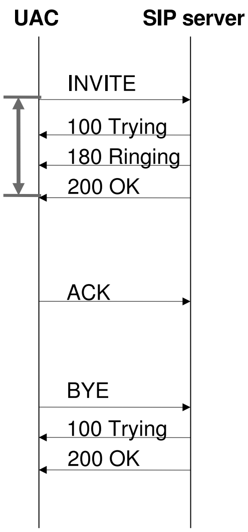

Figure 16: Persistent TCP sequence for INVITE-200 OK test

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 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 drooping the success rate implies that the overload control works ruthlessly for TCP, while it works gracefully for UDP. The overload control at sipd set 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.

5.4 Call Test Scenario: INVITE-200 OK Test

For a call test, UACs and UASes should register their locations beforehand. UACs send INVITE requests, receive 180 Ringing and 200 OK responses from the UASes, and send ACK for the 200 OK. Just after establishing a dialog, the UACs send BYE request to tear down the dialog and receive the 200 OK response. For the call test, we measured the transaction response time of INVITE-200 OK test for three cases of TCP connection lifetimes: transaction-based TCP in Figure 14, dialog-based TCP in Figure 15 and persistent TCP in Figure 16.

To simplify the sequence of the INVITE-200 OK test, we assume zero ringing duration, zero dialog duration, and no mid-dialog request. By this simplification, the three cases differ only in the number of TCP control messages, such as SYN or FIN.

As described in Section 5.1, the SIP server runs as an outbound proxy and also as an inbound proxy. As an outbound proxy, the SIP server operates in passive open and close mode. However, as an inbound proxy, the SIP server opens a connection in active mode, requesting a new connection to the UAS, while it closes TCP connections in passive mode, based on TCP close requests from the UAS. In our configuration, since the UAS closes a TCP connection after each transaction, the SIP server as an inbound proxy operates similarly to the transaction-based TCP. This configuration between the SIP server and the UAS might make unclear the difference among the TCP connection lifetimes between the SIP server and the UAC. However, we could see the largest impact of transaction-based TCP for a SIP server compared to UDP.

5.5 Results of INVITE-200 OK test

Figure 17 shows that the sustainable request rate for all TCP cases is at 700 requests/second and that for UDP is at 900 requests/second. Both sustainable rates are only approximately 17 percent of those for the registration tests, since the call test sequence contains seven times as many SIP messages as the registration sequence, and additional TCP connections between the SIP server and the UAS as compared in Table 1. However, this sustainable request rate, 700 requests/second, is much above the requirement in our traffic model, 167 requests/second in Section 3.2. Also for the requirement including mid-dialog requests, this sustainable rate, which is translated to 1,400 requests/second by adding BYE requests, is above the requirement, 833 requests/seconds.

Figures 18 and 19 show that the success rate, CPU time and memory usage as a function of throughput for the SIP server, for persistent TCP and for UDP, respectively. For persistent TCP, the success rate drops dramatically to seven percent above 700 requests/second. On the other hand, for UDP, the success rate gradually decreased above 900 requests/second. We omit presenting the results for transaction-based and dialog-based TCP, since they are similar to that for persistent TCP. This is probably because the SIP server performs as an inbound proxy for transaction-based TCP for all the three cases.

As we discussed in the results of the registration test, we have determined that this drop of the success rate is not caused by exhausting system resources e.g., CPU or memory, but caused by the overload control at sipd. Figures 18 and 19 shows that CPU utilization is still below 40 percent and virtual memory consumes at most 1,050 MB, which are slightly more than the registration test. When the success rate drops, sipd produced warning messages saying dropping 93 percent of requests by the overload control.

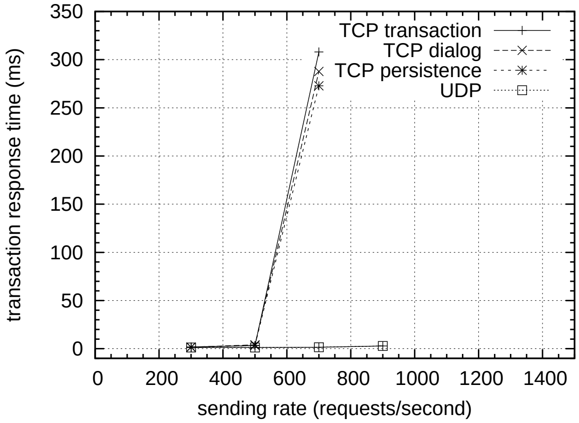

Figure 17: Transaction response times as a function of sending rate for INVITE-200 OK test

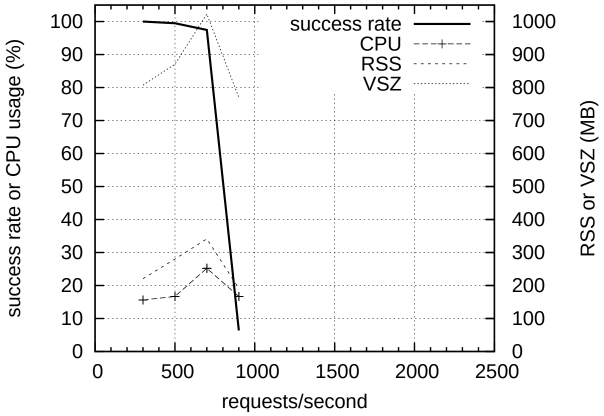

Figure 18: Success rate, CPU utilization and memory usage for INVITE-200 OK test: persistent TCP

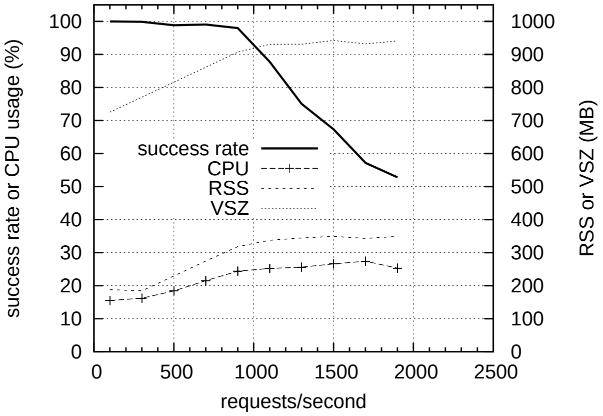

Figure 19: Success rate, CPU utilization and memory usage for INVITE-200 OK test: UDP

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 20 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 butter 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 21, 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.

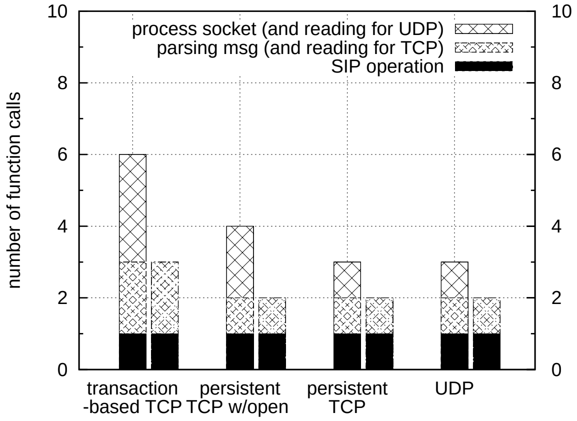

Figure 20: Number of function calls and threads for reading and parsing a REGISTER message

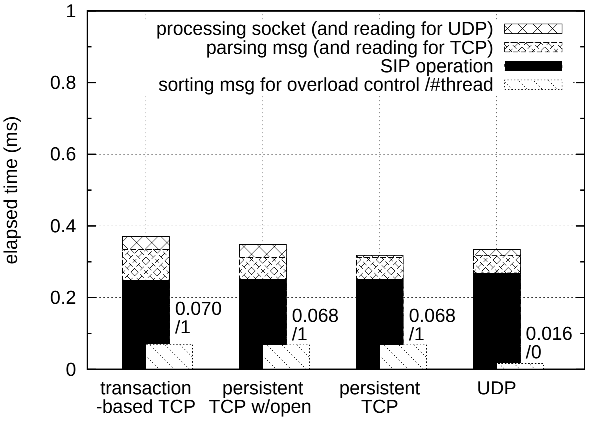

Figure 21: Elapsed time for reading and parsing a REGISTER message

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 environment, 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 Conclusions and Future Work

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. The impact on the response time is not significant under our target traffic model.

However, under heavy loads, e.g., 700 requests/second for the call test, the major impact is on the performance and on the success rate. The response time exponentially increases around the sustainable rate. 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 the SIP server. From the results of the component tests, we suggest to speed up message parsing to ease overload control for a SIP server.

We will measure the impact of choosing SCTP (Stream Control Transmission Protocol), which is defined as a transport protocol for SIP, on server scalability. Since this new transport protocol seems more complicated than TCP, it would be important to clarify the impact of sustainable connections.

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, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, October 2000.

[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-patches/nio-improve.html` (accessed in January, 2006).

[6] K. Shemyak and K. Vehmanen. Scalability of TCP Servers, Handling Persistent Connections. Technical report, 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. 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] H. Schulzrinne. Cinema:sipd. `http://www.cs.columbia.edu/IRT/cinema`.

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

A Comparison of System Calls to Wait for Events

We used the `epoll()` system call to wait for events, instead of the `poll()` or `select()` system call. This is because the `epoll()` system call is the most scalable.

The `select()` system call has an upper limit on the file descriptor set size of 1,024, while user applications increase an upper limit for the `poll()` system call. The `epoll()` system call is more scalable than the `poll()` system call for the following reasons.

Separated interface

Both of `poll()` and `select()` provide a combined interface for setting up the polling list and for retrieving events. The `epoll()` system call separates interfaces for each, `epoll_create()`, `epoll_wait()` and `epoll_ctl()`, so that the system can build efficient data structure in kernel and user spaces.

> The `epoll()` system call is only for Linux, but similar system calls exist for other OS, such as `kqueue()` for FreeBSD or `/dev/poll` for Solaris.

The `epoll_create()` system call builds the persistent data structure for a set of polling file descriptors only in kernel space, while the `poll()` system call builds the interest sets both in kernel and user space.

Efficient event retrieval

After the `epoll_ctl()` system call adds or removes a polling file descriptor from the set, the `epoll_wait()` system call waits for events with preparing a data structure in user space for retrieving events. The data structure in user space is much shorter list than that for `poll()`, since the `epoll_wait()` can set the list only for the number of events to be handled at once.

When an event occurs on a file descriptor, the kernel sets the event at the top of the list, while the kernel using `poll()` sets it corresponding to the prepared file descriptor. Then, a user application scans the list to find the new event from the top. Thus, user applications using `epoll()` can find the event more efficiently than that using `poll()` that needs to scan the full list. Therefore, `epoll()` system calls improves I/O performance.

B System Configurations

B.1 Server Configurations

The following command line increases the number of file descriptors in the system;

```
% echo 1048576 > /proc/sys/fs/file-max
```

1,048,576 (= 1024*1024) is the system limit defined as a constant, `NR_FILE` in `include/linux/fs.h`. To increase this limit when enough memory is installed, we need to modify and recompile the kernel.
To change the `/proc/sys` parameters at boot time, we need to add them to `/etc/sysctl.conf` as follows:

```
fs.file-max=1048576
```

The `ulimit` command can be used to increase the number of file descriptors per process:

```
% ulimit -n 1000000
```

To allow a remote shell to access a large number of file descriptors for our measurement, we need to specify the user name and the parameter in `/etc/security/limits.conf`:

```
special_user       soft   nofile   1000000
special_user       hard   nofile   1000000
```

To allow a remote shell to access a larger number of file descriptors via ssh, we need to restart `sshd` in `/etc/rc.local`:

```
% ulimit -n 1000000
% /etc/rc.d/init.d/sshd restart
```

The memory space for TCP socket buffer is configured as follows:

```
net.ipv4.tcp_rmem = 4096        87380   174760
net.ipv4.tcp_wmem = 4096        16384   131072
net.ipv4.tcp_mem = 98304        131072  196608
```

These parameters are automatically configured at boot time based on available memory as well as TCP established and bind hash table entries. We can see the variables at boot log in `/var/sys/message`:

```
kernel: TCP established hash table entries: 524288 (order: 10, 4194304 bytes)
kernel: TCP bind hash table entries: 65536 (order: 7, 524288 bytes)
kernel: TCP: Hash tables configured (established 524288 bind 65536)
```

B.2 Client Configurations

To increase the number of file descriptors for a shell, we again use the `ulimt` command:

```
% ulimit -n 60000
```

To increase the range of local ports, we modify the `ip_local_port_range` file:

```
% echo 10000   65535 > /proc/sys/net/ipv4/ip_local_port_range
```

C Measurement Tools

We monitored our measurement metrics every second by the following tools. We monitored the “inuse” field to measure the number of TCP connections, and the “mem” field to monitor the allocated pages for TCP socket buffers in `/proc/net/sockstat` file at server. We used the `free` command to measure the total memory usage at server, and `slabinfo` command to measure slab cache. We measured the response time and elapsed time by adding the timestamps at programs, `gettimeofday()`. For accuracy, we subtracted the overhead of calling `gettimeofday()`. To monitor how many resources an application consumes, we measured CPU utilization with the `top` command, and memory usage, RSS (resident set size) and VSZ (virtual memory size), with the `ps` command.