On TCP-based SIP Server Overload Control

Charles Shen and Henning Schulzrinne
Department of Computer Science, Columbia University
{charles, hgs}@cs.columbia.edu
Technical Report CUCS-048-09
November 10, 2009

***Abstract*— SIP server overload management has attracted interest recently as SIP becomes the core signaling protocol for Next Generation Networks. Yet virtually all existing SIP overload control work is focused on SIP-over-UDP, despite the fact that TCP is increasingly seen as the more viable choice of SIP transport. This report answers the following questions: is the existing TCP flow control capable of handling the SIP overload problem? If not, why and how can we make it work? We provide a comprehensive explanation of the default SIP-over-TCP overload behavior through server instrumentation. We also propose and implement novel but simple overload control algorithms without any kernel or protocol level modification. Experimental evaluation shows that with our mechanism the overload performance improves from its original zero throughput to nearly full capacity. Our work leads to the important high level insight that the traditional notion of TCP flow control alone is incapable of managing overload for time-critical session-based applications, which would be applicable not only to SIP, but also to a wide range of other common applications such as database servers.**

I. Introduction

The Session Initiation Protocol (SIP) [48] is an application layer signaling protocol for creating, modifying, and terminating media sessions in the Internet. SIP has been adopted by major standardization bodies including 3GPP, ITU-T, and ETSI as the core signaling protocol of Next Generation Networks (NGN) for services such as VoIP, conferencing, Video on Demand (VoD), presence, and Instant Messaging (IM). The increasingly wide deployment of SIP has raised the requirements for SIP server overload management solutions [47]. SIP server can be overloaded for many reasons such as emergency-induced call volume, flash crowds generated by TV programs (e.g., American Idol), special events such as "free tickets to third caller", or denial of service attacks.

Although SIP server is an application server, the SIP server overload problem is distinct from other well-known application server such as HTTP overload for at least three reasons: First, it is common for a SIP session to traverse multiple hops of SIP proxy servers until it reaches the final destination. This characteristics creates a so-called SIP proxy-to-proxy overload scenario which is absent in the mostly single-hop client-server HTTP architecture. Second, SIP defines a number of application level retransmission timers to deal with possible packet losses, especially when running over an unreliable transport like UDP. This protocol retransmission mechanism can have an adverse effect when the server is overloaded. On the other hand, HTTP is predominantly running over TCP and does not possess the same application layer retransmission problem. Third, SIP requests are much more time sensitive than HTTP requests since SIP signaling is mostly used for real-time sessions.

SIP already has a mechanism that sends rejection messages to terminate sessions that it could not serve. However, one of the key property of SIP server overload is that the cost of rejecting a session is usually comparable to the cost of serving a session. Consequently, when a SIP server has to reject a large number of incoming sessions, it ends up spending all its processing cycles for rejection, causing its throughput to collapses. If, as often recommended, the rejected sessions are sent to a load-sharing SIP server, the alternative server will soon also be generating nothing but rejection messages, leading to a cascading failure.

Since the built-in rejection mechanism is incapable of handling SIP overload, Hilt *et al* [57], [58] articulate a SIP overload control framework based on augmenting the current SIP specification with application level feedback from the SIP Receiving Entity (RE) servers to the SIP Sending Entities (SE) servers. The feedback, which may be rate-based or window-based, could delegate the burden of rejecting excessive calls from the RE to the SE and thus prevent that the RE is being overwhelmed by the SEs. Detailed application level feedback algorithms and their effectiveness for SIP overload control have been demonstrated by a number of researchers, e.g., Noel [40], Shen [55] and Hilt [28].

It is worth pointing out that virtually all existing SIP overload control design and evaluation focus on SIP-over-UDP, presumably because UDP is still the common choice for today's SIP operational environment. However, SIP-over-TCP is getting increasingly popular and seen as a more viable SIP transport choice in the near future for a number of reasons: first, there is a growing demand for securing SIP signaling [46] with the standard SIP over TLS [54] solution, e.g., as mandated by the SIP Forum [1]. TLS itself runs on top of TCP; second, deployment of SIP in NGN is expected to support longer message sizes that exceed the maximum size UDP can handle, forcing carriers to turn to SIP-over-TCP. Third, there are other advantages that TCP holds which motivates a shift to run SIP-over-TCP, such as easier firewall and NATs traversal.

The SIP-over-TCP overload control problem possesses two distinct aspects when compared to the SIP-over-UDP overload control problem. One is TCP's built-in flow control mechanism

which provides an inherent, existing channel for feedback-based overload control. The other is the removal of many application layer retransmission timers that exacerbates the overload condition in SIP-over-UDP. Nahum *et al* [16] have experimentally studied SIP performance and found that upon overload the SIP-over-TCP throughput exhibits a congestion collapse behavior as with SIP-over-UDP. Their focus, however, is not on overload control so they do not discuss why SIP-over-TCP congestion collapse happens or how to prevent it. Hilt *et al* [28] have shown simulation results applying application level feedback control to SIP servers with TCP-specific SIP timers but without including a TCP transport stack in the simulation.

This report systematically addresses the SIP-over-TCP overload control problem. To the authors' knowledge, our paper is the first to provide a comprehensive answer to the following questions: why are there still congestion collapse in SIP-over-TCP despite the presence of the well-known TCP flow control mechanism and much fewer SIP retransmission timers? Is there a way we can utilize the existing TCP infrastructure to solve the overload problem without changing the SIP protocol specification as is needed for the UDP-based application level feedback mechanisms?

We find that the key reasons why TCP flow control feedback does *not* prevent SIP congestion collapse has to do with the session-based nature and real-time setup requirement of SIP load. Request and response messages in the same SIP session arrive at different times from upstream and downstream SIP entities; start-of-session requests trigger all the remaining in-session messages and are therefore especially expensive. The transport level connection-based TCP flow control, without knowing the causal relationship about the messages, will admit too many start-of-session requests and result in a continued accumulation of in-progress sessions in the system. The messages for all the admitted sessions soon fill up the system buffers and entail a long queueing delay. The long delay not only triggers the SIP end-to-end response retransmission timer, but also significantly slows down the effective rate of server session setup. This forms a back pressure through the TCP flow control window feedback which ultimately propagates upstream to the session originators, hindering the session originators from generating further in-session messages that could complete the setup of accepted sessions. The combined delayed message generation and processing as well as response retransmission lead to SIP-over-TCP congestion collapse.

Based on our observations, we propose novel SIP overload control mechanisms within the existing TCP flow control infrastructure. To accommodate the distinction between start-of-session requests and other messages, we introduce the concept of *connection split*. To meet the delay requirements and prevent retransmission, we develop *smart forwarding* algorithms combined with *buffer minimization*. Our mechanism contains only a single tuning parameter for which we provide a recommended value. Implementation of our mechanism exploits existing Linux socket API calls and is extremely simple. It does not require any modifications at the kernel level, neither does it mandates any change to the SIP or TCP specification. We evaluate our mechanism on a common Intel-based Linux test-bed using the popular open source OpenSIPS [44] server with up to ten upstream SEs overloading the RE at up to 10 times the server capacity. The performance is found to be improved from zero to full capacity with our mechanisms. We also show that under heavy overload, the mechanism maintains a fair share of the capacity for competing upstream SEs.

Our research leads to the important insight that the traditional notion of TCP flow control alone is insufficient for preventing congestion collapse for real-time session-based loads, which cover a broad range of applications, e.g., from SIP servers to datacenter systems [59]. Additional techniques, like what we have proposed, are needed and they could be simple but very effective.

The remainder of this paper is structured as follows. Section II describes related work. Section III provides some background on SIP and TCP flow and congestion control. Section IV describes the experimental testbed used for our experiments. Section V explains the SIP-over-TCP congestion collapse behavior. Section VI develops and evaluates our overload control mechanism.

II. Related Work

A. SIP Server Performance and Overload Control

Many researchers have studied SIP server performance. Schulzrinne *et al* presented SIPstone [51], a suite of SIP benchmarks for measuring SIP server performance on common tasks. Cortes [14] measured the performance of four different stateful SIP proxy server implementations over UDP. Nahum *et al* [16], [39], Oho and Schulzrinne [41] showed experimental SIP over UDP and TCP performance results using OpenSER and SIPd SIP server, respectively. Ram *et al* [45] demonstrated that the process architecture in OpenSER causes a substantial performance loss for using SIP over TCP and provided improvements. Salsano *et al* [49] and Camarillo [11] measured the performance of SIP proxy server over UDP, TCP and TLS based on a Java-based proxy implementation and on ns-2 simulator, respectively. While the above work does not specifically study SIP overload control, the results to different extents exhibit the SIP congestion collapse behavior under overload.

SIP overload falls into the broader category of application server overload, which has received extensive study in the area related to HTTP (web) server overload control. Server adaptive QoS management and service differentiation for clients are common techniques proposed for web server overload [2], [4], [17], [42], [61]. Many other researchers combined admission control with service differentiation [8], [20], [32], [35], [63]. Unlike web servers, it is difficult for SIP servers to provide the similar concept of differentiated QoS for different sessions during overload, because the basic task is setting up the call session. Another general method to alleviate the web server overload problem is to adaptively distribute the load across a cluster of web servers [13], [69]. As far as SIP overload

is concerned, drafting additional SIP servers alone does not completely solve the problem.

Although most of the web server overload study like the above uses a request-based workload model, Cherkasova and Phaal [12] presented a study using session-based workload for E-commerce web applications. They proposed several adaptive, self-tunable internal admission control strategies which aimed at minimizing the percentage of aborted requests and refused connections and maximizing the achievable server throughput in completed sessions. The key idea is to monitor the server load periodically and estimate the server capacity. If the load exceeds the estimated capacity, more sessions should be rejected to reduce the load. Since requests that could not be accommodated are explicitly rejected, they considered the rejection cost, but only within the range where the rejection cost is still not high enough to exhaust the server. This assumption would be unrealistic in our SIP server overload study, so we do not make such an assumption.

The SIP server overload problem itself has received intensive attention only recently. Ejzak *et al* [18] provided a qualitative comparison of the overload in PSTN SS7 signaling networks and SIP networks. Whitehead [64] described a protocol-independent overload control framework called GOCAP. But it is not yet clear how exactly SIP can be mapped into the framework. Ohta [36] explored the approach of using a priority queueing and bang-bang type of overload control through simulation. Noel and Johnson [40] presented initial results of a rate-based SIP overload control mechanism. Sun *et al* [56] proposed adding a front end SIP flow management system to conduct overload control including message scheduling, admission control and retransmission removal. Sengar [53] combined the SIP built-in backoff retransmission mechanism with a selective admittance method to provide server-side pushback for overload prevention. Hilt *et al* [28] provided a side-by-side comparison of a number of overload control algorithms for a network of SIP servers, and also examined different overload control paradigms such as local, hop-by-hop and end-to-end overload control. Shen *et al* [55] proposed three new window-based SIP feedback control algorithms and compared them with rate-control algorithms. Most of the above work on SIP overload control assumes UDP as the transport. Hilt *et al* [28] does include simulation of application level feedback overload control for SIP server with only TCP-specific timers enabled, but without a TCP transport stack.

B. TCP Flow and Congestion Control and Its Performance

The basic TCP flow and congestion control mechanisms are documented in [31], [43]. Modifications to the basic TCP algorithm have been proposed to improve various aspects of TCP performance, such as start-up behavior [29], retransmission fast recovery [21], packet loss recovery efficiency [23], [37], or more overall improvements [3], [9]. Research has also been extended to optimize the TCP algorithm for more recent network architecture such as mobile and wireless networks [6], [10], [19], [65], [67] and high-speed networks [26], [33], [66], [68]. There are also efforts focus not on modifying TCP flow and congestion control algorithm itself, but on using dynamic socket buffer tunning methods to improve performance [15], [25], [27], [52]. Another category of related work addresses routers, e.g., active buffer management [22], [38] and router buffer sizing [60]. Our work differs from all the above in that our metrics is not the direct TCP throughput, but the application level throughput. Our goal is to explore the existing TCP flow control mechanism, and to develop a mechanism for a SIP-over-TCP system that neither requires modifying existing TCP algorithm specification, nor needs any kernel level modification such as dynamic socket buffer tuning.

A number of studies have also investigated TCP performance for real-time media [5], [7], [34], [62]. Our work, however, is concerned about the session establishment phase, or the control plane of multimedia real-time services, which has very different load characteristics and usually more constrained latency requirements.

III. Background

A. SIP Overview

SIP defines two basic types of entities: User Agents (UAs) and servers. UAs represent SIP end points. SIP servers consist of registrar servers for location management, and proxy servers for message forwarding. SIP messages are divided into requests (e.g., INVITE and BYE to create and terminate a SIP session, respectively) and responses (e.g., 200 OK for confirming a session setup). The set of messages including a request and all its associated responses is called a SIP transaction.

SIP message forwarding, known as proxying, is a critical function of the SIP infrastructure. This forwarding process is provided by proxy servers and can be either stateless or stateful. Stateless proxy servers do not maintain state information about the SIP session and therefore tend to be more scalable. However, many standard application functionalities, such as authentication, authorization, accounting, and call forking, require the proxy server to operate in a stateful mode by keeping different levels of session state information. Therefore, we focus on stateful SIP proxying.

Figure 1 shows a typical message flow of stateful SIP proxying. Two SIP UAs, designated as User Agent Client (UAC) and User Agent Server (UAS), represent the caller and callee of a multimedia session. The UAC wishes to establish a session with the UAS and sends an INVITE request to proxy A. Proxy A looks up the contact address for the SIP URI of the UAS and, assuming it is available, forwards the message to proxy B, where the UAS can be reached. Both proxy servers also send 100 Trying response to inform the upstream SIP entities that the message has been received. After proxy B forwards the message to the UAS. The UAS acknowledges receipt of the INVITE with a 180 Ringing response and rings the callee's phone. When the callee actually picks up the phone, the UAS sends out a 200 OK response. Both the 180 Ringing and 200 OK make their way back to the UAC. The UAC then generates an ACK request for the 200 OK. Having established the session, the two endpoints

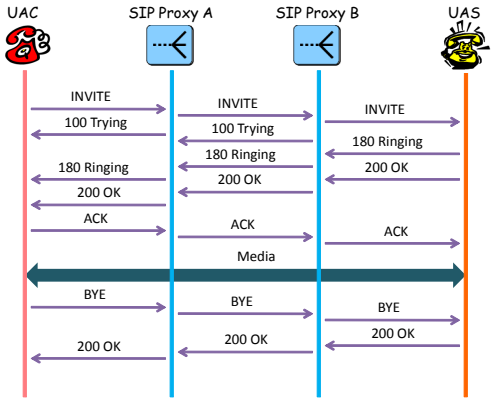


Fig. 1. Basic SIP call flow

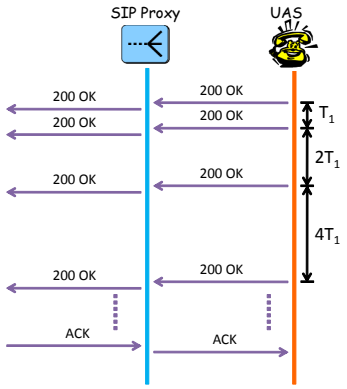
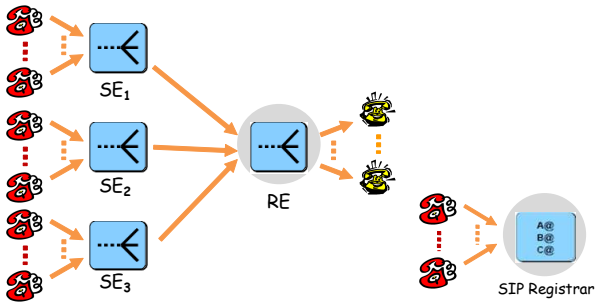


Fig. 2. 200 OK retransmission



(a) proxy to proxy overload (b) UA to registrar overload

Fig. 3. Types of SIP server overload

communicate directly, peer-to-peer, using a media protocol such as RTP [50]. The media session does not traverse the proxies, by design. When the conversation is finished, the UAC "hangs up" and generates a BYE request that the proxy servers forward to the UAS. The UAS then responds with a 200 OK response which is forwarded back to the UAC.

SIP is an application level protocol on top of the transport layer. It can run over any common transport layer protocols, such as UDP, TCP and SCTP. SIP defines quite a number of timers. One group of timers is for hop-to-hop message retransmissions in case a message is lost. These retransmission timers are not used when TCP is the transport because TCP already provides a reliable transfer. There is however a retransmission timer for the end-to-end 200 OK responses which is enabled even when using TCP transport, in order to accommodate circumstances where not all links in the path are using reliable transport. The 200 OK retransmission timer is shown in Fig 2. The timer starts with $T_1 = 500\,ms$ and doubles until it reaches $T_2 = 4\,s$. From then on the timer value remains at T_2 until the total timeout period exceeds 32 s, when the session is considered to have failed. Note that even if the whole path is TCP-based, as long as the message round trip time exceeds 500 ms, the 200 OK timer will expire and trigger retransmission. The UAC should generate an ACK upon receiving a 200 OK. The UAS ceases the 200 OK retransmission timer when it receives a corresponding ACK.

B. Types of SIP Server Overload

There are many causes to SIP overload, but the resulting SIP overload cases can usually be grouped into either of the two types: proxy-to-proxy overload or UA-to-registrar overload.

A typical proxy-to-proxy overload topology is illustrated in Fig 3(a), where the overloaded RE is connected to a relatively small number of upstream SEs. One example of proxy-to-proxy overload is a special event like "free tickets to the third caller", also referred to as flash crowds. Suppose RE is the service provider for a hotline N. SE_1, SE_2 and SE_3 are three service providers that reach the hotline through RE. When the hotline is activated, RE is expected to receive a large call volume to the hotline from SE_1, SE_2 and SE_3 that far exceeds its usual call volume, potentially putting RE into overload.

The second type of overload, known as UA-to-registrar overload, is when a large number of UAs overload the next hop server directly. A typical example is avalanche restart, which happens when power is just restored after a mass power failure in a large metropolitan area and a huge number of SIP devices boot up trying to perform registration simultaneously. This paper only discusses the proxy-to-proxy overload problem.

C. TCP Window-based Flow Control Mechanism and Related Linux API Calls

TCP is a reliable transport protocol with its built-in flow and congestion control mechanisms. Flow control is exercised between two TCP end points. The purpose of TCP flow control is to avoid a sender from sending too much data that overflow the receiver's socket buffer. Flow control is achieved by having the TCP receiver impose a receive window on the sender side indicating how much data the receiver is willing to accept at that moment; on the other hand, congestion control is the process of TCP sender imposing a congestion window by itself to avoid congestion inside the network. The TCP sender assesses network congestion by observing transmission timeout or the receipt of duplicate TCP ACKs, and adjusts the congestion window to slow down or increase the transmission rate as appropriate. Thus, a TCP sender is governed by

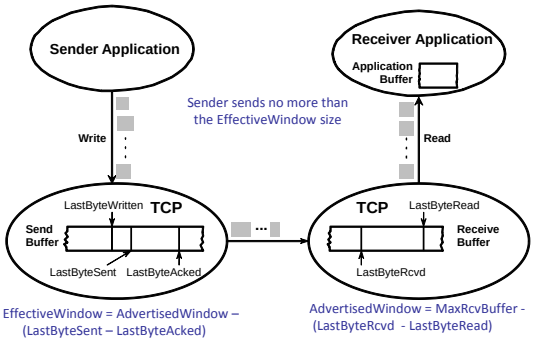


Fig. 4. TCP flow control

both the receiver flow control window and sender congestion control window during its operation.

The focus of our work is on using TCP flow control since we are interested in the receiving end point being able to deliver transport layer feedback to the sending end point and we want to see how it could facilitate higher layer overload control. We illustrate the TCP flow control architecture in Fig 4. A socket level TCP connection usually maintains a send buffer and a receive buffer at the two connection end points. The receiver application reads data from the receive buffer to its application buffer. The receiver TCP computes its current receive buffer availability as its advertised window to the sender TCP. The sender TCP never sends more data than an effective window size derived based on the receiver advertised window and data that has been sent but not yet acknowledged.

In our experimental testbed, the default send buffer size is 16 KB and the default receive buffer size is 85 KB. Since the Linux operating system uses about 1/4 of the socket receive buffer size for bookkeeping overhead, the estimated effective default receive buffer size is about 64 KB. In the rest of the paper we use the effective value to refer to receive buffer sizes. The SIP server application that we use allocates a default 64 KB application buffer.

Linux also provides convenient API to allow applications to manipulate connection-specific socket buffer sizes using the `SO_SNDBUF` and `SO_RCVBUF` options of the `setsockopt` function call. It should be noted that when using `setsockopt` to supply a socket send or receive buffer size, the Linux system doubles the requested size. E.g., if we supply 8 K as `SO_SNDBUF` to `setsockopt`, the system will return a 16 KB send buffer. Furthermore, at the receiver side, if we specify a 1,365 B socket receive buffer, the system doubles its size to allocate a 2,730 B receive buffer. Excluding the 1/4 overhead, the effective receive buffer is then about 2KB.

In addition, Linux supports various API calls that allow the applications to retrieve real-time status information about the underlying TCP connection. For example, using the `SIOCOUTQ` option of the `ioctl` call, the application can learn about the amount of unsent data currently in the socket send buffer.

IV. Experimental Testbed and Metrics

A. Server and Client Software

We evaluate the Open SIP Server (OpenSIPS) version 1.4.2 [44], a freely-available, open source SIP proxy server. OpenSIPS is a fork of OpenSER, which in turn is a fork of SIP Express Router (SER) [30]. These sets of servers represent the *de facto* open source version of SIP server, occupying a role similar to that of Apache for web server. All these SIP servers are written in C language, use standard process-based concurrency with shared memory segments for sharing state, and are considered to be highly efficient. We also implement our overload control mechanisms on the OpenSIPS server.

We choose the widely used open source tool, SIPp [24] (May 28th 2009 release) to generate SIP traffic. We also make corrections to SIPp for our test cases. E.g., we find that existing SIPp implementation does not enable 200 OK retransmission timer over TCP as required by the SIP specification, and therefore we added it.

B. Hardware, Connectivity and OS

The overloaded SIP RE server has 2 Intel Xeon 3.06 GHz processors with 4 GB RAM. However, for our experiments, we only use one processor. We use up to 10 machines for SEs, and up to 10 machines for UACs. All the SE and UAC machines either have 2 Intel Pentium 4 3.00 GHz processors with 1 GB memory or 2 Intel Xeon 3.06 GHz processors and 4 GB RAM. The server and client machines communicate over copper Gigabit or 100Mbit Ethernet. Typical round trip time measured by the `ping` command between the machines is around 0.2 ms. All machines use Ubuntu 8.04 with Linux kernel 2.6.24.

C. Test Suite, Load Pattern and Performance Metrics

We wrote a suite of Perl and Bash scripts to automate running the experiments and analyzing results. Our test load pattern is the same as in Fig 1. For simplicity but without affecting our evaluation purpose, we do not include call holding time and media. That means, the UAC sends a BYE request immediately after sending an ACK request. In addition, we do not consider the time between the ringing and the actual pick-up of the phone. Therefore, the UAS sends a 200 OK response immediately after sending a 180 Ringing response.

Our main performance metrics include server throughput which reflects the per-second number of sessions successfully set up by receiving the ACK to 200 OK at UAS. We also examine Post Dial Delay (PDD), which corresponds to the time from sending the first INVITE to receiving the 200 OK response. A number of other metrics such as CPU utilization and server internal message processing rate are also used in explaining the results.

V. Default SIP Over TCP Overload Performance

We start our evaluation with a single SE - single RE testbed with all out-of-the-box configurations and show the throughput in Fig. 5. It can be seen that the throughput immediately collapses as the load approaches and exceeds the

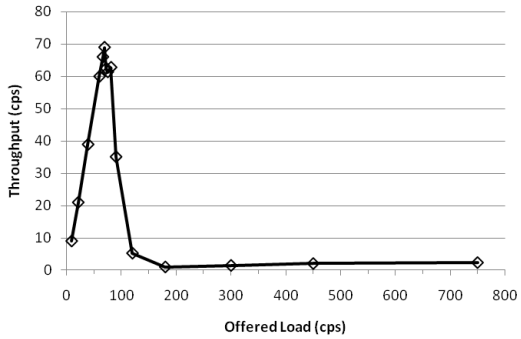


Fig. 5. Default SIP-over-TCP throughput

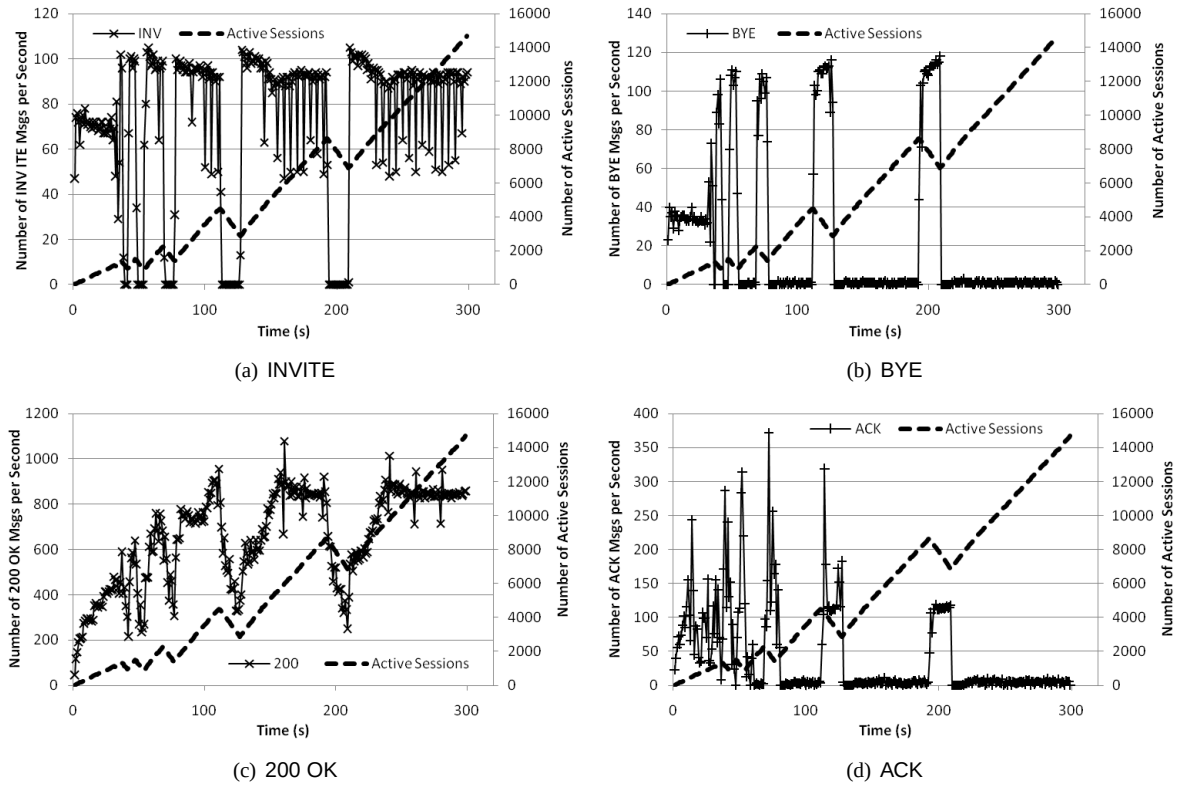
server capacity. In this section, we explore the detailed causes of this behavior through server instrumentation.

We examine a particular run at a load of 150 cps which is about 2.5 times the server capacity. Fig. 6 depicts the per second message processing rate. The four figures show INVITE, BYE, 200 OK and ACK, respectively. It should be noted that the number of 180 Ringings, not shown in these figures, basically follows the number of INVITEs processed, because the UAS is not overloaded and can always deliver responses to RE. For the same reason, the number of 200 OKs to BYEs which are also not shown, follow the number of BYEs. Along with the individual message processing rates, Fig. 6 also includes the current number of active sessions in the RE. The active sessions are those sessions that have been started by an INVITE but have not yet received a BYE. Since the call holding time is zero, in an ideal situation, any started sessions should be terminated immediately, leaving no session outstanding in the system. In a real system, the number of active sessions could be greater than zero. The larger the number of such in-progress sessions, the longer the delay that those sessions will experience.

Fig. 6 indicates that 200 OK retransmission happens almost immediately as the test starts, which means the end-to-end round trip delay immediately exceeds 500 ms. This is caused by the large buffers at the different stages of the network system, which allow too many sessions to be accepted. The SIP session load is not atomic. The INVITE request is always first introduced into the system and then come the responses and follow-up ACK and BYE requests. When too many INVITEs are admitted to the system, the BYE generation rate cannot keep up with the INVITEs, resulting in a large number of active sessions in the system and also a large number of messages queued in various stages of the buffers. These situations translate to prolonged delays in getting the ACK to 200 OK to the UAS. More specifically, assuming the server's capacity is 65 cps, if the sessions are indeed atomic, each session will take a processing time of 15.4 ms. In order to avoid 200 OK retransmission, the end-to-end one-way delay cannot exceed 250 ms, corresponding to a maximum of about 16 active sessions in the system. Factoring in the non-atomic nature of the session load, this maximum limit could be roughly doubled to 32. But with the default system configuration, we have a 16 KB TCP socket send buffer, and 64 KB socket receive buffer, as well as 64 KB SIP server application buffer. Considering an INVITE size of around 1 KB, this configuration means the RE can be filled with up to 130 INVITEs at one time, much larger than the threshold of 32. All these INVITEs contribute to active sessions once admitted. In the experiment, we see the number of active sessions reaches 49 at second 2, immediately causing 200 OK retransmissions. 200 OK retransmissions also trigger re-generated ACKs, adding more traffic to the network. This is why during the first half of the time period in Fig. 6, the number of ACKs processed is higher than the number of INVITEs and BYEs processed. Eventually the RE has accumulated too many INVITEs both in its receive buffer and application buffer. So its flow control mechanism starts to advertise a zero window to the SE, blocking the SE from sending additional INVITE requests. Subsequently the SE stops processing INVITE requests because of the send block to the RE. This causes SE's own TCP socket receive buffer and send buffer to get full as well. The SE's flow control mechanism then starts to advertise a zero window to UAC. This back pressure on UAC prevents the UAC from sending anything out to the SE. Specifically, the UAC can neither generate new INVITE requests, nor generate more ACK and BYEs, but it could still receive responses. When this situation happens, retransmitted 200 OKs received can no longer trigger retransmitted ACKs. Therefore, the number of ACKs processed in the later half of the graph does not exceed the number of INVITEs or BYEs. The number of ACKs becomes actually similar to the number of BYEs because BYEs and ACKs are generated together at the same time in our workload.

It can further be seen that under the default settings, the INVITE and BYE processing tends to alternate with gradually increasing periods as the test proceeds. During each period, the INVITE portion is increasingly larger than the BYE portion. Since the number of active sessions always increases with INVITE processing, and decreases with BYE processing, those processing patterns lead to the continued growth of the number of active sessions in the RE and exacerbate the situation.

In addition to observing the per-second message processing rate at RE, we also confirm the behavior from the total number of messages processed at the UAS, along with the number of active sessions at RE as in Fig. 7. Note that the number of INVITEs received, 180 Ringing and initial 200 OK (not retransmissions) messages sent are the same, because 180 Ringing and 200 OK are generated by UAS immediately upon receiving an INVITE. Similarly the number of ACK, BYE, and 200 OK to BYEs are the same, because ACK and BYE are generated at the same time at the UAC and 200 OK to BYE is immediately generated upon receiving BYE at the UAS. In Fig. 7, initially between 0 and the 38th second, the number of ACK and BYEs received are roughly half of the total INVITEs received. Therefore, the number of active sessions in the RE and the number of ACKs received at the UAS are roughly the same. Then RE enters the abnormal INVITE processing and



(a) INVITE (b) BYE

(c) 200 OK (d) ACK

Fig. 6. RE message processing rates and number of active sessions in default SIP-over-TCP test

BYE processing alternating cycle. During the period when RE is processing ACKs and BYEs, the number of active sessions decreases. During the period when RE is processing INVITEs, no ACKs are forwarded, so the number of ACKs remains constant.

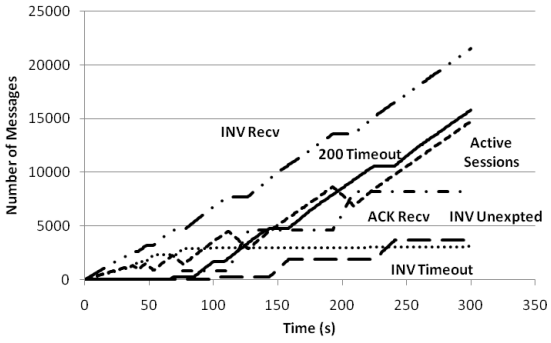


Fig. 7. Total number of messages processed at UAS and number of active sessions at RE

200 OK retransmission starts at second 2. The total period of 200 OK retransmission lasts 32 seconds for each individual session, therefore the expiration of the first session that has exhausted all its 200 OK retransmissions without receiving an ACK happens at the 34th second. The actual 200 OK retransmission timeout we see from Fig. 7 is at the 66th second. The difference between the 66th and 34th second is 32 seconds, which is a configured maximum period UAS waits to receive the next message in sequence, in this case the ACK to 200 OK.

Starting from the 69th second, we see a category of messages called *INVITE Unexpected*. These are indeed ACKs and BYEs that arrive after the admitted sessions have already timed out at the UAS.[1]These ACKs and BYEs without a matching session also create session states at the SIPp UAS, which normally expect a session message sequence beginning with an INVITE. Since those session states will not receive other normal in-session messages, at the 101th second, or after the 32 seconds UAS receive timeout period, those session states start to time out, reflected in the figure as the *INVITE Timeout* curve. Finally, a very important overall observation from Fig. 7 is that at a certain point, the 77th second, the number of timely received ACKs virtually stopped growing, causing the throughput to drop to zero.

We also show the final screen logs at the UAC and UAS side for the test with default configurations in Fig. 8, where status code 202 is used instead of 200 to differentiate the 200 OK to BYE from the 200 OK to INVITE. We have explained the 200 OK retransmissions, 200 OK timeouts, INVITE timeouts, and INVITEs unexpected messages. We can see that among the

[1]Note that the number of active sessions still sees a decrease although those processed BYEs are for sessions that have expired, this is because the RE active session statistics merely records the difference between the total number of INVITEs and BYEs processed without taking delay into consideration.

		Messages	Retrans	Timeout	Unexpected-Msg
INVITE ---------->	B-RTD1	25899	0		
100 <----------		25899	0	0	0
477 <----------		0	0	0	0
180 <----------		25899	0	0	0
200 <----------	E-RTD1	25899	216652	0	0
ACK ---------->		10106	0		
BYE ---------->	B-RTD2	10106	0		
202 <----------	E-RTD2	3821	0	6285	3567

(a) UAC

		Messages	Retrans	Timeout	Unexpected-Msg
----------> INVITE	B-RTD1	25899	0	6285	13576
<---------- 180		25899	0		
<---------- 200		25899	223167	22078	
----------> ACK	E-RTD1	3821	0	0	0
----------> BYE		3821	0	0	0
<---------- 202		3821	0		

(b) UAS

Fig. 8. Screen logs in default SIP-over-TCP test

25,899 INVITEs received at the UAS side, 22,078 eventually time out and only 3,821 receive the final ACK. The UAC actually sends out a total of 10,106 ACKs and BYEs. The remaining 6,285 ACKs and BYEs are indeed delivered to UAS but are too late when they arrive, therefore those BYEs do not trigger 202 OK and we see 6,285 202 OK timeouts at the UAC. At the UAS side, those 6,285 ACKs and BYEs establish abnormal session states and eventually time out after the 32 s receive timeout for INVITE. The unexpected messages at the UAC side are 408 Send Timeout messages triggered at the SIP servers for the BYEs that do not hear a 202 OK back. Note that the number of those messages (3,567) is smaller than the exact number of BYEs that do not receive 202 OK (6,285). This is because the remaining 2,718 408 Send Timeout messages arrive after the 202 OK receive timeout and therefore those messages were simply discarded and not counted in the screen log.

We also examine the PDD in Fig. 9. Even if we do not consider whether the ACK are delivered to complete session setup, the results show that 73% of the INVITEs have a PDD between 8 and 16 seconds, which is most likely beyond the human interface acceptability limit. Another 24% have a PDD between 4 to 8 seconds, which might be at the boundary of the acceptable limit.

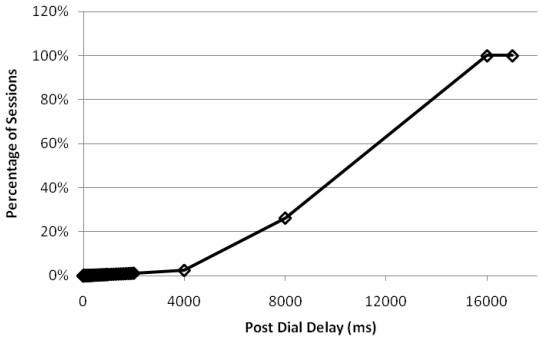


Fig. 9. PDD in default SIP-over-TCP test

VI. SIP-over-TCP Overload Control Mechanism Design

Key lessons we learn from SIP-over-TCP congestion collapse is that we must limit the number of INVITEs we can admit to avoid too many active sessions accumulating in the system, and for all admitted INVITEs we need to make sure the rest of the session messages complete within finite delay. In this section, we propose specific approaches to address these issues, namely *connection split*, *buffer minimization*, as well as *smart forwarding*.

A. Connection Split and Buffer Minimization

First, it is clear that we only want to limit INVITEs but not non-INVITEs because we do not want to drop messages for sessions already accepted. In order to have a separate control of INVITEs and non-INVITE messages, we split the TCP connection from SE to RE into two, one for INVITE requests, and the other for all other requests. Second, in order to limit the number of INVITEs in the system and minimize delay, we minimize the total system buffer size between the SE and the RE for the INVITE connection, which should include three parts: the SE TCP socket send buffer, the RE TCP socket receive buffer and the RE SIP server application buffer. We call the resulting mechanism *Explicit Connection Split + Buffer Minimization* (ECS+BM) and illustrate it in Fig. 10.

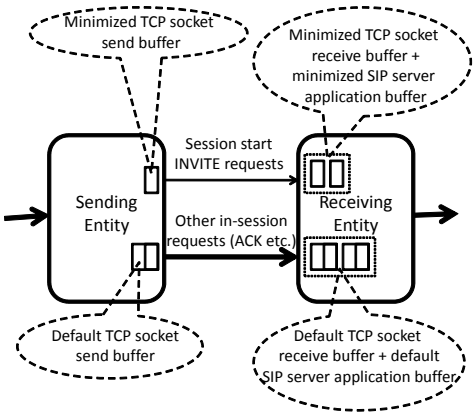


Fig. 10. Explicit Connection Split + Buffer Minimization

We find, however, although ECS+BM effectively limits the number of INVITEs that could accumulate at the RE, the resulting throughput differs no much from that of the default configuration. The reason is that, since the number of INVITEs SE receives from UAC remains the same and the INVITE buffer sizes between SE and RE are minimized, the INVITE pressure merely moves a stage back and accumulates at the UAC-facing buffers of the SE. Once those buffers, including the SE receive buffer and SE SIP server application buffer, have been quickly filled up, the system delay dramatically increases. Furthermore, UAC is then blocked from sending to SE and unable to generate ACKs and BYEs, causing the number of active sessions in the RE to skyrocket.

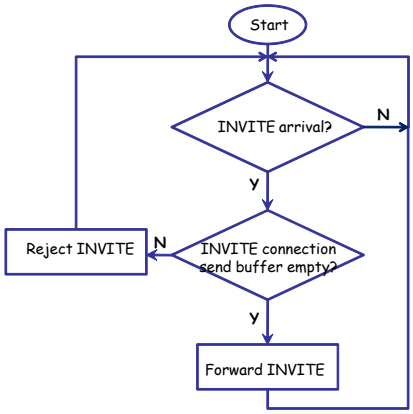


Fig. 11. Smart forwarding for ECS

B. Smart Forwarding

In order to release, rather than pushing back, the excessive load pressure present in the ECS+BM mechanism, we introduce the *Smart Forwarding* (SF) algorithm as shown in Fig. 11. This algorithm is enforced only for the INVITE connection. When an INVITE arrives, the system checks whether the current INVITE connection send buffer is empty. If yes, the INVITE is forwarded; otherwise the INVITE is rejected with an explicit SIP rejection message. This algorithm has two special advantages: first, although we can choose any send buffer length threshold value for rejecting an INVITE, the decision to use the emptiness criterion makes the algorithm parameter-free; second, implementation of this algorithm is especially easy in Linux systems because the current send buffer occupancy can be retrieved by a simple standard `ioctl` call.

C. Explicit Connection Split, Buffer Minimization and Smart Forwarding (ECS+BM+SF)

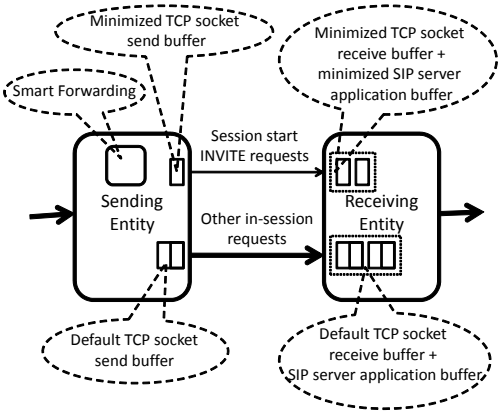


Fig. 12. ECS+BM+SF illustration

Our resulting mechanism is then ECS+BM+SF and we illustrate it in Fig. 12. Basically, RE listens to two separate sockets, one for INVITE requests that start new sessions, the other for other in-session requests, such as ACKs and BYEs. SIP response messages go through the reverse directions of the corresponding connection as usual. We start with the following settings for the special INVITE request connection: the SE send buffer size is set to the minimum system-allowed value of 2 KB; the RE side effective TCP socket receive buffer is set to about 1 KB and the RE application buffer size is set to 1,200 bytes. Since the size of an INVITE in our test is about 1K, these configurations allows the RE to hold at maximum one or two active INVITEs at a time.

We compare the detailed results of this ECS+BM+SF mechanism with those of the default configuration in the same scenario as Section V with one SE overloading an RE at an offered load of 2.5 times the capacity.

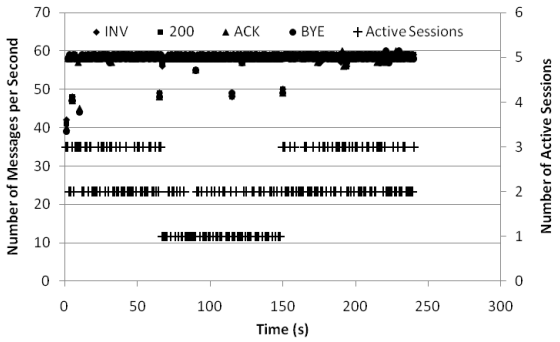


Fig. 13. RE message processing rates with ECS+MB+SF

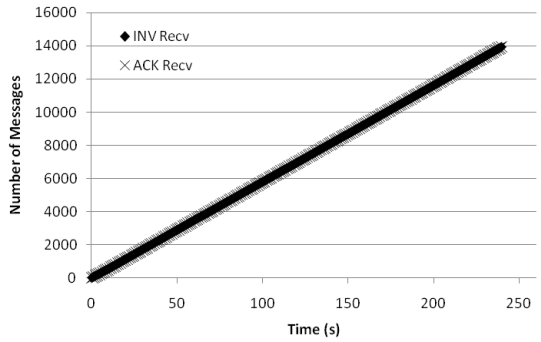


Fig. 14. UAS total number of message processing with ECS+MB+SF

Fig. 13 shows the average message processing rate and the number of active sessions in the RE. We can see dramatic difference of this figure from Fig. 6. Here, the values of INVITE, 200 OK, ACK, and BYE processing rate overlap most of the time, which explains why the number of active sessions remains extremely low, between 1 and 3, all the time. Fig. 14 shows that the total numbers of INVITEs and ACKs received at the UAS are consistent. The slope of these two overlapping lines corresponds to the throughput seen at the UAS[2].

[2]The throughput value, 58 cps, is smaller than the peak value in Fig. 5 at the same load because this run is obtained with substantial debugging code enabled.

```
                          Messages  Retrans   Timeout   Unexpected-Msg
INVITE ---------->  B-RTD1 35999     0
   100 <----------         35999     0         0         0
   477 <----------         22019     0         0         0
   180 <----------         13980     0         0         0
   200 <----------  E-RTD1 13980     0         0         0
   ACK ---------->         13980     0
   BYE ---------->  B-RTD2 13980     0
   202 <----------  E-RTD2 13980     0         0         0
```

(a) UAC

```
                          Messages  Retrans   Timeout   Unexpected-Msg
----------> INVITE      B-RTD1 13980     0         0         0

<---------- 180               13980     0
<---------- 200               13980     0         0
----------> ACK         E-RTD1 13980     0         0         0

----------> BYE               13980     0         0         0
<---------- 202               13980     0
```

(b) UAS

Fig. 15. Screen logs with ECS+MB+SF

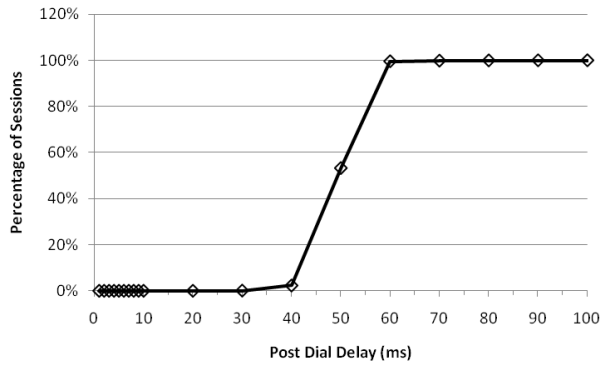


Fig. 16. PDD with ECS+MB+SF

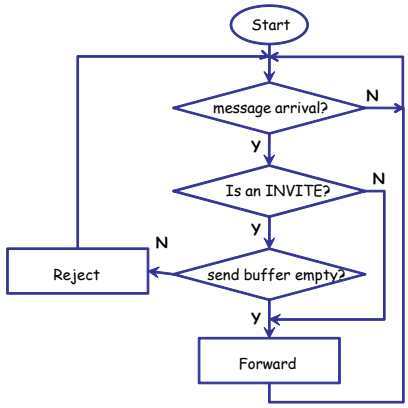


Fig. 17. Smart forwarding for ICS

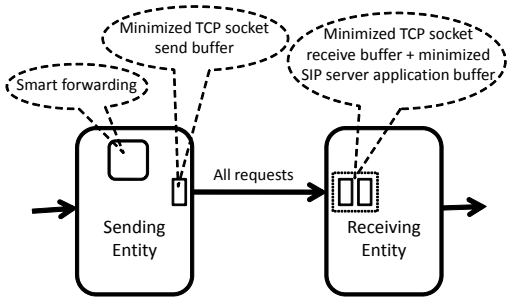


Fig. 18. ICS+BM+SF illustration

The PDD of the test is shown in Fig. 16. As can be seen, none of the delay values exceeds 700 ms, and over 99% of the sessions has a delay smaller than 60 ms. Furthermore, from the overall UAC and UAS screen logs in Fig. 15(a) and Fig. 15(b), we see that among the 35,999 INVITEs that are generated, 22,019 of them are rejected by the *smart forwarding* algorithm. The remaining 13,980 sessions all successfully get through, without triggering any retransmission or unexpected messages - a sharp contrast to Fig. 8. Finally, the system achieves full capacity as confirmed by the full CPU utilization observed at the RE.

D. Implicit Connection Split, Buffer Minimization and Smart Forwarding (ICS+BM+SF)

The ECS+BM+SF mechanism in Section VI-C is effective in restricting load by combining *smart forwarding* and two separate connections for INVITE and non-INVITE requests, with special buffer minimization techniques applied to the INVITE connection. If the mechanism works so well in keeping only a few active sessions in the RE all the time, we deduce that servers should never be backlogged and therefore the queue size for both INVITE and non-INVITE request connections should be close to zero. In that case, the dedicated connection for non-INVITE requests does not require the default large buffer setting either. We may therefore merge the two split connections back into one with a minimized SE send buffer, RE receive and application buffer settings. We also need to revise the *smart forwarding* algorithm accordingly as in Fig. 17. Since there is only a single request connection now, the algorithm checks for INVITE requests and rejects it if the send buffer is non-empty. Otherwise, the INVITE is forwarded. All non-INVITE requests are always forwarded. Although the revised mechanism no longer requires a dedicated connection for INVITEs, it treats INVITEs and non-INVITEs differently. Therefore, we call it *Implicit Connection Split* (ICS) as opposed to the previous ECS. We show the resulting ICS+BM+SF mechanism in Fig. 18. Running the same overload experiment as in Section VI-C, we see that the RE average message processing rate Fig. 19 and UAS total message processing Fig. 20 are pretty similar to Fig. 13 and Fig. 14.

However, the number of active sessions in the system is between 0 to 3 in ICS as opposed to between 1 to 3 in ECS. This indicates that the ICS mechanism is more conservative in forwarding INVITEs (or more aggressive in rejecting INVITEs) because in ICS INVITEs and non-INVITEs share a single connection and the same buffer space. This will imply that ICS could have a smaller delay but also smaller throughput than ECS. Fig. 21 compares the PDD of ICS and ECS. In ICS, over 99.8% of the sessions have a delay value smaller than 30 ms, much better than ECS where 99% of the session delays are smaller than 60 ms. On the other hand, Fig. 22 shows that ICS successfully admitted 13,257 of the 35,999 INVITEs, only

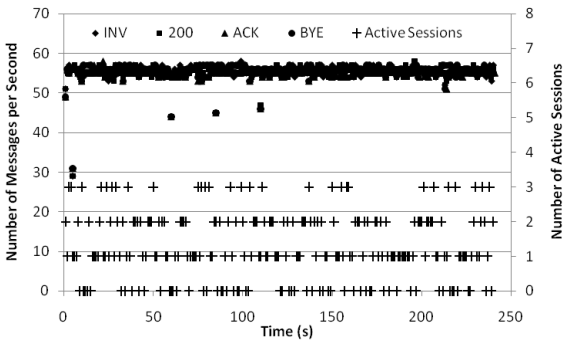


Fig. 19. RE message processing rates with ICS+MB+SF

an insignificant 5% fewer than the corresponding number in ECS. Combining with the big advantage of not requiring an explicit connection split, these results indicate ICS could be a more preferable choice over ECS during overload.

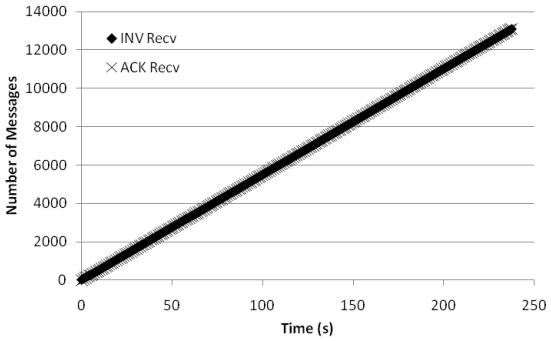


Fig. 20. UAS total number of message processing with ICS+MB+SF

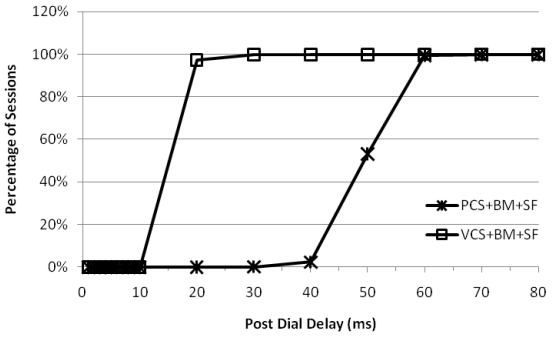


Fig. 21. PDD with ICS+MB+SF vs. ECS+MB+SF

VII. SIP-over-TCP Overload Control Mechanism Parameter Tuning

The mechanisms developed in Section VI contain three tuning parameters which are the three buffer sizes. We minimized their values and set the SE send buffer to 2 KB, RE receive buffer to 1 KB and RE application buffer to 1,200 bytes. In this section we explore the relationship among setting different values of these three buffer sizes.

```
                                Messages  Retrans   Timeout   Unexpected-Msg
INVITE ----------> B-RTD1 35999     0
   100 <----------          35999     0         0         0
   477 <----------          22742     0         0         0
   180 <----------          13257     0         0         0
   200 <----------  E-RTD1  13257     0         0         0
   ACK ---------->          13257     0
   BYE ----------> B-RTD2  13257     0
   202 <----------  E-RTD2  13257     0         0         0
```

(a) UAC

```
                                Messages  Retrans   Timeout   Unexpected-Msg
----------> INVITE     B-RTD1 13257     0         0         0

<---------- 180              13257     0
<---------- 200              13257     0         0
----------> ACK        E-RTD1 13257     0         0         0

----------> BYE              13257     0         0         0
<---------- 202              13257     0
```

(b) UAS

Fig. 22. Screen logs with ICS+MB+SF

A. Increasing the RE Side Buffer

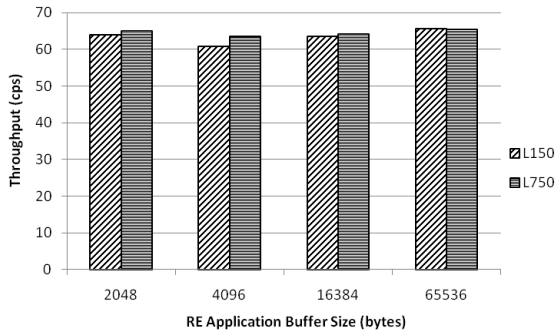
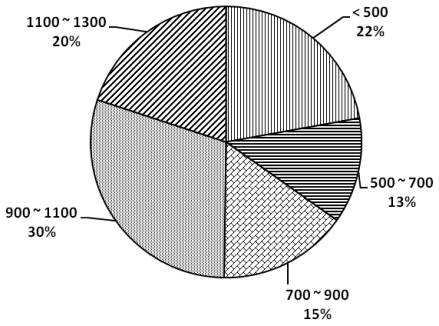


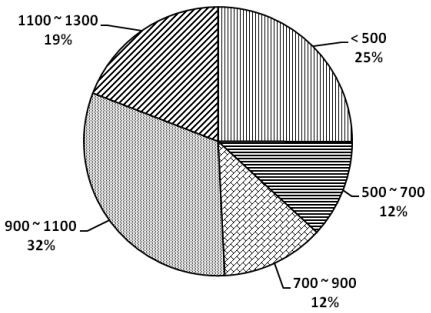
Fig. 23. Throughput under varying RE application buffer with minimized SE send buffer and RE receive buffer

1) Increasing Either RE Application Buffer or Receive Buffer: First we keep the SE send buffer and RE receive buffer size at their minimized values, and see how increasing the RE application buffer may affect performance. We specifically look at the throughput under two load values, 150 cps and 750 cps, the former representing a moderate overload of 2.5 times the capacity and the latter a heavy overload of 12.5 times capacity. The application buffer sizes vary at 2 KB, 4 KB, 16 KB, 64 KB. The 64 KB value is the default application buffer size. Fig. 23 shows that the application buffer size does not have a noticeable impact on the throughput. Moreover, the number of 200 OK retransmissions is found to be zero in all the tests, indicating a timely completion of all the session setup.

To further illustrate the actual sizes of application buffer used, we plot the histograms of actual number of bytes RE reads in each time from the receive buffer in two tests: with minimized send buffer and receive buffer but default application buffer under load 150 cps and 750 cps. Results in Fig. 24 show that even when the application buffer size is 64 KB, the system almost never reads more than 1,300 bytes. This can be explained by the fact that the number of bytes the application buffer reads are limited by the receive buffer



(a) L=150 cps



(b) L=750 cps

Fig. 24. RE application buffer reading histogram (in Bytes)

TABLE I
MESSAGE SIZES OBSERVED AT SE AND RE (IN BYTES)

Message Type	At SE	At RE
INVITE	776	941
100 Trying	363	NA
180 Ringing	473	534
200 OK	638	699
ACK	425	590
BYE	608	773
202 OK	356	417
Total	2863	3954

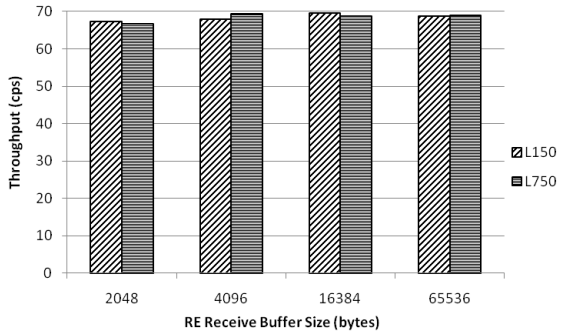


Fig. 25. Throughput under varying RE receive buffer with minimized SE send buffer and RE application buffer

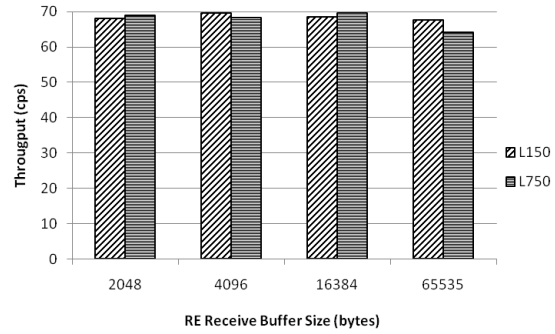
size. Note that in these tests, although the estimated effective receive buffer size is 1 KB, the maximum receive buffer size could be up to 1,360 bytes depending on the actual buffer overhead.

By referring to the message sizes captured by Wireshark at the RE and SE as listed in Table I[3] and check the server message log, we confirm that most of the time, the bytes read are for a single or a couple of messages which are sent together. E.g., since the 180 Ringing and 200 OK messages are sent at the same time, they are likely to be read together, which account for about 1,233 bytes. Therefore, a larger RE application buffer size actually does not change throughput once the other two buffers are already minimized.

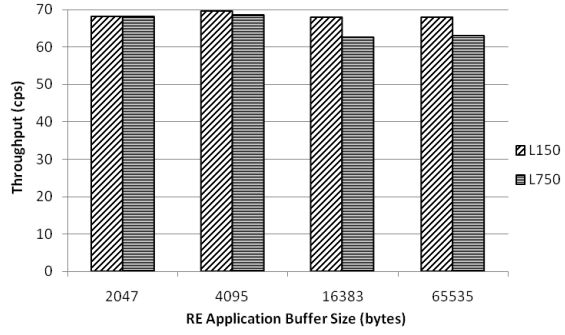
Results in Fig. 25 indicate that when the send buffer and application buffer are minimized, the throughput does not make a difference even when the receive buffer is increased up to its 64 KB default value.

2) Increasing Both RE Receive Buffer and Application Buffer: We have known from Section VII-A1 that keeping either of the RE receive buffer or RE application buffer at its default value, while minimizing the other one still works. Can the minimized RE receive buffer or RE application buffer be further increased while the other one is in its default value? As Fig. 26 shows, the throughputs do remain close to the system capacity at both heavy and moderate overloads even in those cases.

[3]The differences between the lengths seen at the SE and RE are caused by the serve stripping away or appending certain SIP headers, e.g., the `Route` and `Record-Route` headers.



(a) Varying RE receive buffer with minimized SE send buffer and default RE application buffer



(b) Varying RE application buffer with minimized SE send buffer and default RE receive buffer

Fig. 26. Increasing both RE receive buffer and application buffer

However, recall that enlarging either RE buffer size could hold messages in the RE and increase queuing delay. We plot the PDD distribution for four test cases in Fig. 27. Two of those cases compare the delay when RE application buffer is set to 2 KB vs. the default 64 KB, while the RE receive buffer is at its default value of 64 KB. Most of the delays in the small application buffer case are below 375 ms, and as a result we observe no 200 OK retransmissions at the UAS side. In the large application buffer case, however, nearly 70% of the sessions experience a PDD between 8 seconds and 32 seconds, which will most likely be hung up by the caller even if the session setup messages could ultimately complete. Not surprisingly, we also see a large number of 200 OK retransmissions in this case.

The other two cases in Fig. 27 compare the PDD when the receive buffer is set to 2 KB vs. the default 64 KB, while the application buffer is at its default value of 64 KB. In the small receive buffer case, over 99.7% of the sessions have a PDD below 30 ms, and there is certainly no 200 OK retransmissions at the UAS side. In the larger receive buffer case, about 30% of the sessions have a PDD below 480 ms, and the rest 70% between 480 ms and 700 ms. Since a large number of sessions experienced a round trip delay exceeding 500 ms, we see quite a number of 200 OK retransmissions at the UAS side, too.

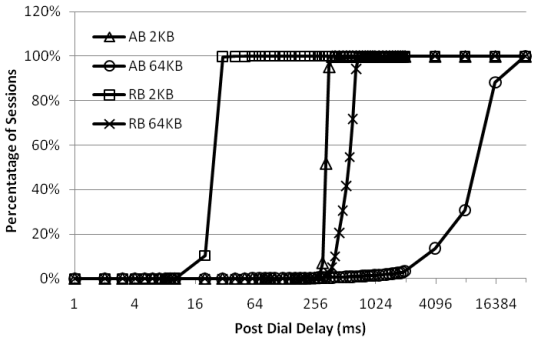
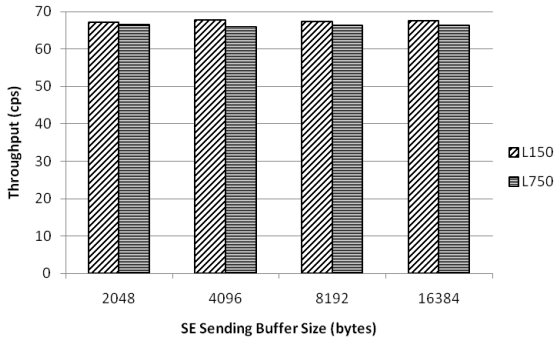
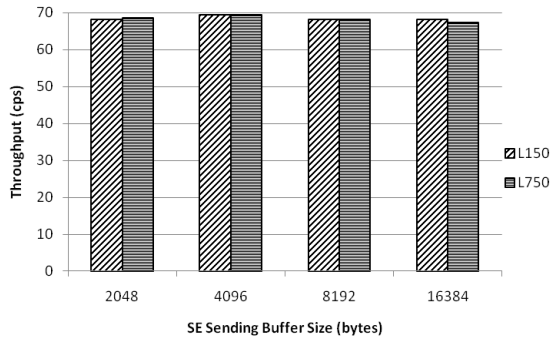


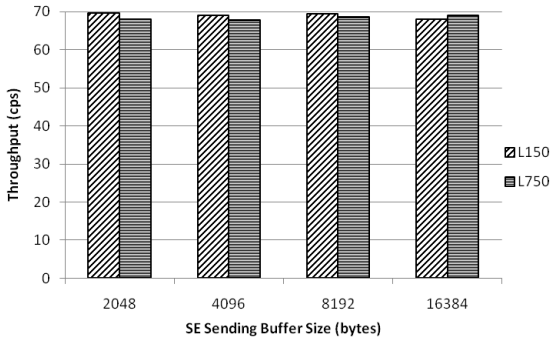
Fig. 27. PDD comparison for RE side buffer tuning



(a) 2 KB receive buffer and 1,300 B application buffer



(b) 2 KB receive buffer and default 64 KB application buffer



(c) Default 64 KB receive buffer and 1,300 B application buffer

Fig. 28. Throughput performance under varying SE sending buffer sizes

To summarize, although throughput is similar by tuning either RE receive buffer or application buffer, the delay performance could be very different in these two approaches. Specifically, when similar size of RE receive buffer or application buffer is used and the other buffer left at its default value, limiting the receive buffer could produce over a magnitude lower PDDs than limiting the application buffer, which in turn significantly reduces the likelihood of 200 OK message retransmissions. The above results suggest that since RE receive buffer and application buffer are connected in series, at least one of them has to be tuned in order to restrict buffering delay, and tuning the receive buffer is preferable over tuning the application buffer. This conclusion also matches the intuition: limiting the receive buffer produces more timely transport feedback than limiting the application buffer.

B. Increasing SE Side Buffer

So far we have explored RE receive buffer and application buffer sizes based on the assumption that SE send buffer is always minimized. In this section we examine the impact of varying SE send buffer size as well. Fig. 28 show the overload throughput again at loads of 150 cps and 750 cps under three different combined RE receive buffer and application buffer settings: both buffers minimized, only receive buffer minimized and only the application buffer minimized. We see that the throughput values in all cases are reasonably close to the system capacity and do not exhibit noticeable differences.

To get a better understanding, we inspect the actually used SE send buffer size in a test run with load 750 cps, default SE send buffer, default RE application buffer and 2 KB RE receive

buffer. Fig. 29 shows the histogram of number of unsent bytes in the SE send buffer when an INVITE arrives but sees a non-empty send buffer. It shows that during over 61% of the times when an INVITE is rejected, the send buffer size is less than 1,000 bytes; during over 99.9% of the times when an INVITE is rejected, the send buffer size is less than 3,000 bytes; the upper bound of number of unsent bytes seen by a rejected INVITE is 5,216 bytes. Furthermore, the number of active sessions at both the SE and RE are found to be within the range of 0 to 4. These numbers are pretty reasonable considering the total length of non-INVITE messages for each session, which is 2,087 bytes as listed in Table I.

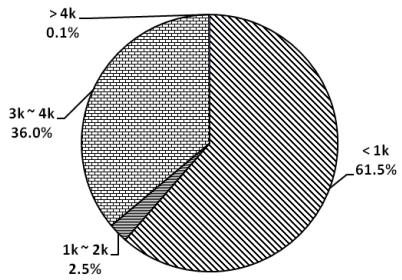


Fig. 29. SE send buffer unsent bytes histogram

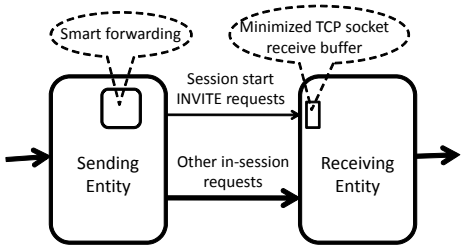
Fig. 29 tells us that the SE send buffer size again does not have to be minimized. This can be attributed to our *smart forwarding* algorithm which essentially prevents excessive non-INVITE message built up in the system. Combined with a minimized buffer at the RE, our mechanism minimizes the number of active sessions in the system, which means there will always be only a small number of messages in the SE send buffer.

VIII. Overall Performance of our SIP-over-TCP Overload Control Mechanisms

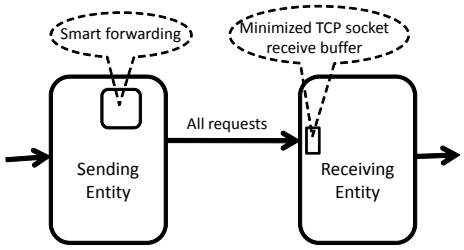
We develop our overload control algorithms in Section VI with only the RE receive buffer as its tuning parameters. The simplified mechanisms are shown in Fig. 30. In this section we evaluate the overall performance of these mechanisms. To demonstrate scalability, we test on three scenarios with 1 SE, 3 SEs and 10 SEs, respectively.

A. Overall Throughput and PDD

Fig. 31 illustrates the throughput with and without our control mechanismsin three test scenarios with varying number of SEs and an offered load up to over 10 times the capacity. The RE receive buffer was set to 2 KB and the SE send buffer and RE application buffer remain at their default values. As we can see, in all test runs with our control mechanisms, the overload throughput maintains at close to the server capacity, even in the most constrained case with 10 SEs and 750 cps. There are subtle differences between ECS and ICS though, as we mentioned in Section VI-D, that ICS is more effective in rejecting sessions than ECS. As a result, although we observe occurrence of 200 OK retransmissions at the 10 SE,

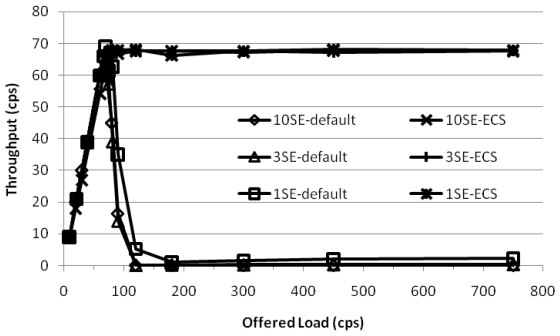


(a) ECS+BM+SF

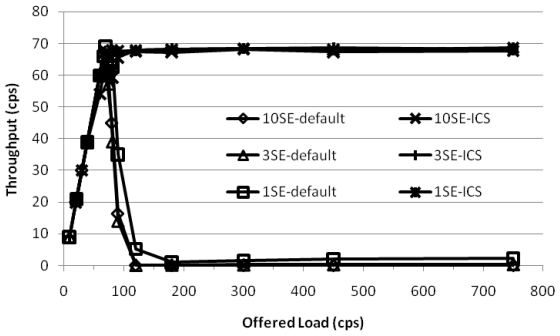


(b) ICS+BM+SF

Fig. 30. SIP-over-TCP overload control mechanisms after parameter simplification



(a) ECS+BM+SF



(b) ICS+BM+SF

Fig. 31. Overall throughput of SIP-over-TCP: with and without our overload control mechanisms

750 cps overload test in ECS, there is no single 200 OK retransmissions in any ICS test runs.

We further compare the ICS tests with different number of SEs. Fig. 32 shows that the numbers of active sessions in

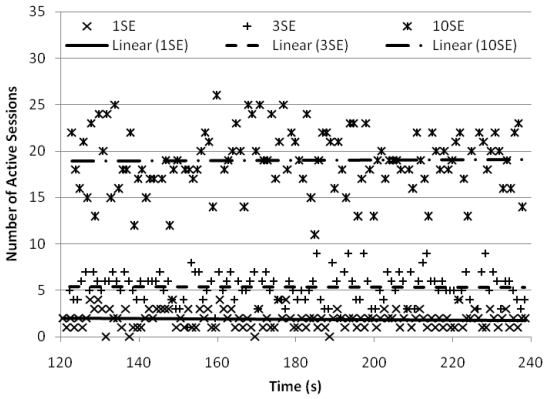


Fig. 32. Number of active sessions in RE in scenarios with varying number of SEs

RE for the three scenarios roughly correspond to the ratio of the numbers of SEs (1:3:10), as would be expected because in our testbed configuration each SE creates a new connection to the RE which will be allocated a new set of RE buffers. Increased number of active sessions causes longer PDDs, as demonstrated in Fig. 33, where the overall trend and the 50 percentile values match the 1:3:10 ratio pretty well.

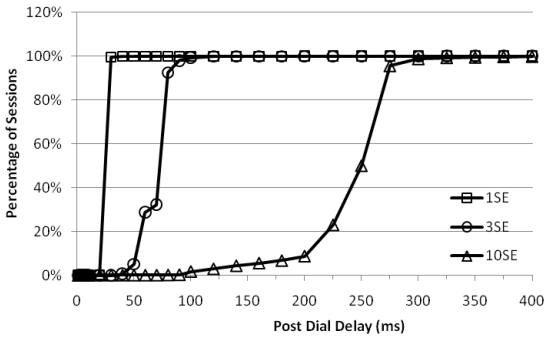


Fig. 33. PDD in scenarios with varying number of SEs

Fig. 32 and Fig. 33 also imply that if the number of SEs keeps increasing until a very large number, we will eventually still see an undesirably large number of active sessions in the system. The PDD will also exceed the response retransmission timer value, although the adverse effect of response retransmission on the actual performance will likely only be observable when the number of such retransmissions accumulates to a certain extent, because the 500 ms retransmission timer value is smaller than the normally several-second acceptable PDD limit, and the processing cost of 200 OK responses is usually not the most expensive among all the messages in the session. The actual crossing point depends on the processing power of the server.

Thus, our mechanism is most applicable to cases where the number of SEs are reasonably small, which however, does cover a fairly common set of realistic SIP server overload scenarios. For example, there are typical national service providers deploying totally hundreds of core proxy and edge proxy servers in a hierarchical manner. The resulting server connection architecture leaves each single server with a few to dozens of upstream servers.

The other cases where a huge number of SEs overloading an RE can occur, e.g., when numerous enterprises, each having their own SIP servers, connect to the same server of a big provider. Deploying our mechanism in those cases will still benefit the performance, but the degree of effectiveness is inherently constrained by the per-connection TCP flow control mechanism itself. Since each SE adds to the number of connections and subsequently to the total size of allocated connection buffers at the RE. As the buffer size accumulates, so does the delay. Indeed, the solution to this numerous-SE-single-RE overload problem may ultimately require a shift from the current push-based model to a poll-based model. Specifically, instead of allowing all the SEs to send, the RE may advertise a zero TCP window to most of the SEs and open the windows only for those SEs that the RE is currently polling to accept loads.

B. RE Receive Buffer Tuning

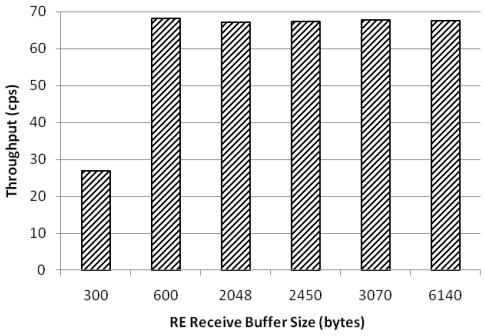


Fig. 34. Impact of RE receive buffer size on Throughput

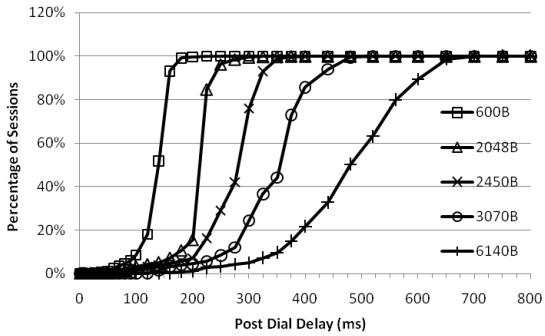


Fig. 35. Impact of RE receive buffer size on PDD

The only tuning parameter in our mechanism is the RE

receive buffer size. We explore the impact of this parameter under the most constrained 10 SEs with load 750 cps case for ICS+MB+SF in Fig. 34. It is not surprising that the receive buffer size cannot be too small because it will cause a single message to be sent and read in multiple segments. After exceeding a certain threshold, the receive buffer does not make difference in overload throughput, but the smaller the buffer is, the lower the PDD, as shown in Fig. 35. The PDD is roughly the same as round trip delay. If the round trip delay exceeds 500 ms, we will start to see 200 OK retransmissions, as in the cases where the receive buffer is larger than 3,070 bytes.

Overload control algorithms are meant to kick in when overload occurs. In practice, a desirable feature is to require no explicit threshold detection about when the overload control algorithm should be activated, because that always introduces additional complexity, delay and inaccuracy. If we keep our overload control mechanism on regardless of the load, then we should also consider how our mechanism could affect the system *underload* performance. We find that in general both ECS and ICS have a pretty satisfactory underload performance, meaning the throughput matches closely with a below-capacity offered load such as in Fig. 31, but comparatively ECS's underload performance is better than ICS because ICS tends to be more conservative. We do observe the ICS mechanism underload throughput noticeably fall below the offered load in a few circumstances, specifically when there is only a single SE, with a receive buffer set around or smaller than the size of a single INVITE, and the load is around 80% to full system capacity. But the combination of these conditions only represents corner cases, which can also be fixed with appropriate parameter tuning if warranted.

Overall, in order to scale to as many SEs as possible yet minimizing the PDD, we recommend an RE receive buffer size that holds roughly a couple of INVITEs.

C. Fairness

All our above tests with multiple SEs assume each SE receiving the same request rate from respective UACs, in which case the throughput for each UAC is the same. Now we look at the situation where each SE receives different request rates, and measure the fairness property of the achieved throughput.

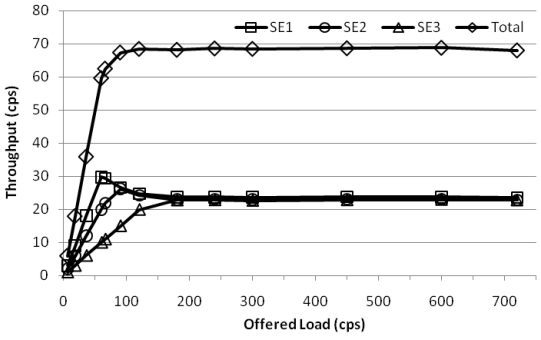


Fig. 36. Throughput: three SEs with incoming load ratio 3:2:1

Fig. 36 shows the throughput of a 3 SE configuration with the incoming offered load to the three SEs distributed at a 3:2:1 ratio. As we can see, when the load is below total system capacity, the individual throughputs via each SE follow the offered load at the same 3:2:1 ratio closely. At light to moderate overload until 300 cps, the higher load sources have some advantages in competing RE resources. At higher overload above 300 cps, each SE receives a load that is close to or higher than the server capacity. The advantages of the relatively higher load SEs are wearing out, and the three SEs basically deliver the same throughputs to their corresponding UACs.

Shen *et al* [55] define two types of fairness for SIP server overload: *service provider-centric* fairness and *end-user-centric* fairness. The former allocates the same portion of the overloaded server capacity to each upstream server; the latter allocates the overloaded server capacity in proportion to the upstream servers' original incoming load. Our results show that the system achieves *service provider-centric* fairness at heavy overload. Obtaining *end user-centric* fairness during overload is usually more complicated, some techniques are discussed in [55].

D. Additional Discussions

During our work with OpenSIPS, we also discover subtle software implementation flaws or configuration guidelines. For example, an SE could block on sending to an overloaded RE. Thus, if there are new requests coming from the same upstream source to the SE which are destined to other REs that are not overloaded, those new requests cannot be accepted either because of the blocking. This is clearly a flaw that will not easily be noticed unless we conduct systematic TCP overload tests. Another issue is related to the OpenSIPs process configuration. OpenSIPS employs a multi-process architecture and the number of child processes is configurable. Earlier work [54] with OpenSIPS has found that configuring one child process yields an equal or higher maximum throughput than configuring multiple child processes. However, in this study we find that when overloaded, the existing OpenSIPS implementation running over TCP with a single child process configuration could lead to a deadlock situation between the sending and receiving entity servers. Therefore, we use multiple child processes for this study.

IX. Conclusions

We experimentally evaluate default SIP-over-TCP overload performance using a popular open source SIP server implementation on a typical Intel-based Linux testbed. Through server instrumentation, we found that TCP flow control feedback cannot not prevent SIP overload congestion collapse because of lack of application context awareness at the transport layer for session-based load with real-time requirements. We develop novel mechanisms that effectively use existing TCP flow control to aid SIP application level overload control. Our mechanism has three components: the first is *connection split* which brings a degree of application level awareness

to the transport layer; the second is a parameter-free *smart forwarding* algorithm to release the excessive load at the sending server before they reach the receiving server; the third is minimization of the essential TCP flow control buffer - the socket receive buffer, to both enable timely feedback and avoid long queueing delay. Implementation of our mechanisms is extremely simple without requiring any kernel or protocol level modification. Our mechanisms work best for the SIP overload scenarios commonly seen in core networks, where a small to moderate number of sending servers may simultaneously overload a receiving server. E.g., we demonstrate the performance improvement from zero to full capacity in our testbed containing up to 10 SEs at over 10 times overload. We also note that scenarios more likely occur at the edge networks, where there are a huge number of SEs overloading one RE, essentially require a solution which shifts from the current push-based model to a poll-based model. Future work is needed in this area.

Our study sheds lights both at software level and conceptual level. At the software level, we discover implementation flaws for overload management that would not be noticed without conducting systematic overload study, even though our evaluated SIP server is a mature open source server. At the conceptual level, our results suggest an augmentation to the long-held notion of TCP flow control: the traditional TCP flow-control alone is incapable of handling SIP-like time-sensitive session-based application overload. The conclusion may be generalized to a much broader application space that share similar load characteristics, such as database systems. Our proposed combined techniques including *connection split*, *smart forwarding* and *buffer minimization* are key elements to make TCP flow control actually work for managing overload of such applications.

X. Acknowledgement

Funding for this work is provided by NTT. The authors would like to thank Arata Koike of NTT, Erich Nahum of IBM Research, and Volker Hilt of Bell Labs/Alcatel-Lucent for helpful discussions.

References

[1] SIP forum. http://www.sipforum.org.

[2] T. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1563–1577, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control . RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[4] J. Almeida, M. Dabu, and P. Cao. Providing differentiated levels of service in web content hosting. In *In First Workshop on Internet Server Performance*, pages 91–102, 1998.

[5] A. Argyriou. Real-time and rate-distortion optimized video streaming with TCP. *Image Commun.*, 22(4):374–388, 2007.

[6] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, page 136, Washington, DC, USA, 1995. IEEE Computer Society.

[7] S. Baset, E. Brosh, V. Misra, D. Rubenstein, and H. Schulzrinne. Understanding the behavior of TCP for real-time CBR workloads. In *Proc. ACM CoNEXT '06*, pages 1–2, New York, NY, USA, 2006. ACM.

[8] N. Bhatti and R. Friedrich. Web server support for tiered services. *Network, IEEE*, 13(5):64–71, Sep/Oct 1999.

[9] L.S. Brakmo and L.L. Peterson. TCP vegas: end to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, Oct 1995.

[10] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *SIGCOMM Comput. Commun. Rev.*, 27(5):19–43, 1997.

[11] G. Camarillo, R. Kantola, and H. Schulzrinne. Evaluation of transport protocols for the session initiation protocol. *Network, IEEE*, 17(5):40–46, Sept.-Oct. 2003.

[12] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Comput.*, 51(6):669–685, 2002.

[13] M. Colajanni, V. Cardellini, and P. Yu. Dynamic load balancing in geographically distributed heterogeneous web servers. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[14] M. Cortes, J. Ensor, and J. Esteban. On SIP performance. *IEEE Network*, 9(3):155–172, Nov 2004.

[15] T. Dunigan, M. Mathis, and B. Tierney. A TCP tuning daemon. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[16] E. Nahum and J. Tracey and C. Wright. Evaluating SIP server performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 349–350, June 2007.

[17] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web*, 2(3):133–142, 1999.

[18] R. Ejzak, C. Florkey, and R. Hemmeter. Network overload and congestion: A comparison of isup and SIP. *Bell Labs Technical Journal*, 9(3):173–182, November 2004.

[19] H. Elaarag. Improving TCP performance over mobile networks. *ACM Comput. Surv.*, 34(3):357–374, 2002.

[20] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pages 276–286, New York, NY, USA, 2004. ACM.

[21] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004.

[22] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, Aug 1993.

[23] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[24] R. Gayraud and O. Jacques. SIPp. http://sipp.sourceforge.net.

[25] Y. Guo, Y. Hiranaka, and T. Akatsuka. Autonomic buffer control of web proxy server. In *WWCA '98: Proceedings of the Second International Conference on Worldwide Computing and Its Applications*, pages 428–438, London, UK, 1998. Springer-Verlag.

[26] S. Ha, I. Rhee, and L. Xu. Cubic: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, 2008.

[27] G. Hasegawa, T. Terai, T. Okamoto, and M. Murata. Scalable socket buffer tuning for high-performance web servers. In *Network Protocols, 2001. Ninth International Conference on*, pages 281–289, Nov. 2001.

[28] V. Hilt and I. Widjaja. Controlling overload in networks of SIP servers. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 83–93, Oct. 2008.

[29] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 270–280, New York, NY, USA, 1996. ACM.

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

[31] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[32] A. Kamra, V. Misra, and E.M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 47–56, June 2004.

[33] D. Kliazovich, F. Granelli, and D. Miorandi. Logarithmic window increase for TCP westwood+ for improvement in high speed, long distance networks. *Comput. Netw.*, 52(12):2395–2410, 2008.

[34] C. Krasic, K. Li, and J. Walpole. The case for streaming multimedia with TCP. In *IDMS '01: Proceedings of the 8th International Workshop on Interactive Distributed Multimedia Systems*, pages 213–218, London, UK, 2001. Springer-Verlag.

[35] K. Li and S. Jamin. A measurement-based admission-controlled web server. *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:651–659 vol.2, 2000.

[36] M. Ohta. Overload Protection in a SIP Signaling Network. In *International Conference on Internet Surveillance and Protection*, 2006.

[37] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, October 1996.

[38] R. Morris. Scalable TCP congestion control. In *Proc. IEEE INFOCOM 2000*, volume 3, pages 1176–1183 vol.3, Mar 2000.

[39] E. Nahum, J. Tracey, and C. Wright. Evaluating SIP proxy server performance. In *17th International Workshop on Networking and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Urbana-Champaign, Illinois, USA, June 2007.

[40] E. Noel and C. Johnson. Initial simulation results that analyze SIP based VoIP networks under overload. In *ITC*, pages 54–64, 2007.

[41] K. Ono and H. Schulzrinne. One server per city: Using TCP for very large SIP servers. In *LNCS: Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, volume 5310/2008, pages 133–148, Oct 2008.

[42] R. Pandey, J. Fritz Barnes, and R. Olsson. Supporting quality of service in http servers. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 247–256, New York, NY, USA, 1998. ACM.

[43] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[44] The OpenSIPS Project. http://www.opensips.org.

[45] K. Kumar Ram, I. Fedeli, A. Cox, and S. Rixner. Explaining the impact of network transport protocols on SIP proxy performance. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 75–84, Texas, USA, April 2008.

[46] Light Reading. VoIP security: Vendors prepare for the inevitable. *VoIP Services Insider*, 5(1), January 2009.

[47] J. Rosenberg. Requirements for Management of Overload in the Session Initiation Protocol. RFC 5390 (Informational), December 2008.

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

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

[50] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

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

[52] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 315–323, New York, NY, USA, 1998. ACM.

[53] H. Sengar. Overloading vulnerability of voip networks. In *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 419–428, 29 2009-July 2 2009.

[54] C. Shen, E. Nahum, H. Schulzrinne, and C.P. Wright. The impact of TLS on SIP server performance. Technical Report CUCS-022-09, Columbia University Department of Computer Science, May 2009.

[55] C. Shen, H. Schulzrinne, and E. Nahum. Session Initiation Protocol (SIP) server overload control: Design and evaluation. In *LNCS: Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, volume 5310/2008, pages 149–173, Oct 2008.

[56] J. Sun, J. Hu, R. Tian, and B. Yang. Flow management for SIP application servers. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 646–652, June 2007.

[57] V. Hilt, E. Noel, C. Shen, and A. Abdelal. Design Considerations for Session Initiation Protocol (SIP) Overload Control. Internet draft, 2009. Work in progress.

[58] V. Hilt and H. Schulzrinne. Session Initiation Protocol (SIP) Overload Control. Internet draft, October 2009. Work in progress.

[59] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 303–314, New York, 2009. ACM.

[60] A. Vishwanath, V. Sivaraman, and M. Thottan. Perspectives on router buffer sizing: recent results and open problems. *SIGCOMM Comput. Commun. Rev.*, 39(2):34–39, 2009.

[61] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 189–202, Berkeley, CA, USA, 2001. USENIX Association.

[62] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: an analytic performance study. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 908–915, New York, NY, USA, 2004. ACM.

[63] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[64] M. Whitehead. GOCAP - one standardised overload control for next generation networks. *BT Technology Journal*, 23(1):144–153, 2005.

[65] X. Wu, M. Chan, and A. Ananda. Improving TCP performance in heterogeneous mobile environments by exploiting the explicit cooperation between server and mobile host. *Comput. Netw.*, 52(16):3062–3074, 2008.

[66] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE Infocom*. IEEE, 2004.

[67] R. Yavatkar and N. Bhagawat. Improving end-to-end performance of TCP over mobile internetworks. In *WMCSA '94: Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 146–152, Washington, DC, USA, 1994. IEEE Computer Society.

[68] T. Yoshino, Y. Sugawara, K. Inagami, J. Tamatsukuri, M. Inaba, and K. Hiraki. Performance optimization of TCP/ip over 10 gigabit ethernet by precise instrumentation. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

[69] W. Zhao and H. Schulzrinne. Enabling on-demand query result caching in dotslash for handling web hotspots effectively. *HOTWEB '06. 1st IEEE Workshop on*, pages 1–12, Nov. 2006.