

Controlling Overload in Networks of SIP Servers

Volker Hilt* and Indra Widjaja†

*Bell Labs, Alcatel-Lucent
Holmdel, NJ 07733, USA

†Bell Labs, Alcatel-Lucent
Murray Hill, NJ 07974, USA

Abstract—The Session Initiation Protocol (SIP) is rapidly being adopted as the signaling protocol for establishing, modifying and terminating multimedia sessions. With the increasing use of SIP in large deployments, it is now becoming apparent that the current SIP design does not easily scale up to large network sizes and SIP servers are not well equipped to handle overload conditions. When a SIP server is operating close to or above its capacity limit, message retransmissions by various SIP timers can cause the network to be severely overloaded and result in an extremely low goodput. In this paper, we first provide a detailed analysis of the behavior of SIP servers under overload. We show that SIP servers are often unable to recover from congestion collapse once it has occurred and that overload can spread throughout a network of SIP servers. We then discuss mechanisms and algorithms for controlling overload in these servers. We found that performing overload control locally at a server provides a simple remedy for light cases of overload; however, it is ineffective in handling higher amounts of load. Finally, we investigate distributed overload control mechanisms for SIP and show that they are effective in controlling overload of SIP servers.

I. INTRODUCTION

The Session Initiation Protocol (SIP) [16] has been very successful in recent years and has become the main signaling protocol for multimedia sessions in the Internet and IP telephony. It serves as a foundation for many of today's Internet-based communication services including Voice-over-IP, instant messaging and presence. The SIP protocol has also been adopted by 3GPP [1] as the basis for its IMS architecture and many telephony service providers are starting to use or are in the process of deploying SIP-based networks.

Despite its popularity and rapidly growing deployments, overload control for the SIP protocol is still little understood. While the SIP protocol has a basic overload control mechanism [16], this mechanism has proven to be ineffective in practice and often cannot prevent SIP server overload. This problem is now becoming increasingly apparent as SIP deployments are reaching large scales and serve a quickly growing number of users.

Overload of a SIP server occurs if the message arrival rate to the server exceeds its message processing capacity. Under overload, the throughput of a SIP server can drop significantly and can reach zero or a small fraction of the server's original capacity. In this case, the server enters into a congestion collapse. In addition to the messages a SIP server receives from external sources, it also generates messages internally, for example, to retransmit a request if a response

has not been received in time. These internal messages add to the overall message arrival rate of the server. A server can therefore be overloaded even at times when the external message arrival rate is less than its processing capacity. This aspect is important as overload in one server, which will become significantly less responsive, can lead to overload in its neighbors. This way, overload can spread in a network of SIP servers and eventually bring down the entire network.

Overload in a network of SIP servers can be triggered by various reasons. It can be the result of many users trying to initiate a SIP call around the same time, for example, when voting for a TV show or in an emergency situation. Another common reason for overload is the failure of a component in a SIP server farm, which degrades the overall processing capacity and requires the remaining servers to keep up with the incoming load. Yet another reason for overload is when many endpoints recover from a failure at the same time and simultaneously register to the network, for example, after a large power outage or a misconfiguration by the service provider [15]. Overload can occur even if a service provider has carefully dimensioned its SIP servers to user demands since the amount of signaling messages generated by SIP endpoints can exceed the average load by orders of magnitude [6]. As for Web servers and existing telephony networks, it is not economical and often not possible to dimension SIP servers for the worst case.

Since overload in SIP servers cannot fully be avoided, it is crucial to equip the SIP protocol with a mechanism that can effectively manage overload [15]. To achieve this goal, it is important to fully understand the behavior of a network of SIP servers under overload. In this paper, we investigate the effects of SIP server overload. Our results reveal that the current mechanisms provided by the SIP protocol cannot prevent congestion collapse. Furthermore, congestion collapse in one server can spread throughout a network and a SIP server in state of congestion collapse cannot easily recover. We show that local overload control techniques can alleviate the problem of overload only in light cases of overload. We propose distributed overload control mechanisms for SIP and show that they are effective in preventing congestion collapse.

After the introduction in Section I and the discussion on related work in Section II, we introduce key aspects of SIP in Section III. Section IV describes our simulation model, Section V provides a detailed performance evaluation of the SIP protocol, and Section VI evaluates the performance of local

overload control. We investigate mechanisms for distributed overload control in Section VII, and conclude the paper in Section VIII.

II. RELATED WORK

The general topic of overload control in communication networks has received a lot of attention in the past. For example, analysis of overload control based on M/M/1 queuing systems has been studied in [4], [12] and Chapter 11 of [18]. In the early 1990s, researchers were also attracted to overload control in telecommunications networks and the Signaling System (SS7) (e.g., [20], [10] and [17]).

A large body of work exist on congestion control for the TCP protocol, including [2], [13], [7] and [14]. While TCP congestion control aims at controlling a flow of many packets from a single sender to a receiver, SIP overload control is focused on individual requests sent from many senders to many receivers through a set of intermediary SIP servers. Since each sender only contributes single requests, end-to-end flow control is limited in its effectiveness. SIP also uses a different retransmission mechanism than TCP. Overload control has also been investigated for HTTP servers, e.g., in [21] and [22] with a focus on including additional servers to offload traffic during periods of overload and local overload control mechanisms. We investigate local overload control for SIP in Section VI.

Overload control for the SIP protocol has received little attention so far. The SIP specification provides a limited overload control mechanism, which we analyze in detail in Section V. The use of this mechanism with a bang-bang control algorithm is described in [11]. Ejzak et al. [5] articulate the need for overload control in SIP and discuss qualitative similarities and differences between ISUP/SS7 and SIP networks. However, they do not provide quantitative results of how overload would affect SIP performance or propose specific approaches on how to handle overload in SIP networks.

GOCAP [6] takes a different approach by defining an abstract, protocol-independent framework for overload control. Even though it is argued that GOCAP can be applied to SIP, a mapping has not yet been defined. Since GOCAP is protocol independent, it uses a single overload control algorithm and its own control feedback loop that is applied to all controlled protocols.

III. SIP BACKGROUND

A. SIP Protocol Overview

SIP is a request/response-based protocol. End users are represented by user agents (UAs), which take the role of a user agent client (UAC) or user agent server (UAS) for a request/response pair. A UAC creates a SIP request and sends it to a UAS. On its way, a SIP request typically traverses one or more SIP servers, also called SIP proxies. The main purpose of a SIP server is to route a request one hop closer to its destination. Responses trace back the path the request has taken.

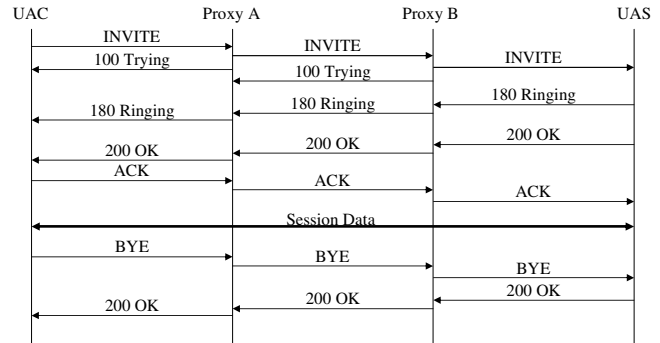


Fig. 1. Example INVITE-BYE call flow.

To set up a SIP session, the UAC sends an INVITE request to the UAS, as shown in Figure 1. Each server on the path confirms the reception of this request by returning a 100 Trying response to the previous hop. Instead of forwarding a request, a SIP server can reject it if it is unwilling or unable to forward the request. Once the request is received by the UAS, it typically responds with a 180 Ringing response to indicate that the called user is being alerted and a 200 OK response when the user has accepted the session. After the 200 OK is received by the UAC, it sends an ACK request to complete the three way handshake of an INVITE transaction. The INVITE request is the only SIP request that uses a three way handshake. Sessions can be terminated at any time by sending a BYE request, which is confirmed with a 200 OK response. The SIP protocol defines other methods besides the ones described above. Since setting up and terminating a session is one of the key usages of SIP, we focus on the call flow for this scenario. We expect the qualitative results to be the same if the other methods are used more frequently in the traffic mix and leave a detailed investigation of the impact of traffic mixes on overload control for further study.

The SIP protocol provides a basic overload control mechanism through the 503 (Service Unavailable) response code. SIP servers that are unable to forward a request due to temporary overload can reject the request with a 503 response. The overloaded server can insert a Retry-After header into the 503 response, which defines the number of seconds during which this server does not want to receive any further request from the upstream neighbor. A server that receives a 503 response from a downstream neighbor stops forwarding requests to this neighbor for the specified amount of time and starts again after this time is over. Without a Retry-After header, a 503 response only affects the current request and all other requests can still be forwarded to this downstream neighbor. A server that has received a 503 response can try to re-send the request to an alternate server, if one is available. A server does not forward 503 responses towards the UAC and converts them to 500 (Server Internal Error) responses instead.

The SIP protocol can operate over reliable (e.g., TCP) and unreliable transport protocols (e.g., UDP). If it runs over an unreliable transport, SIP uses retransmission timers with an

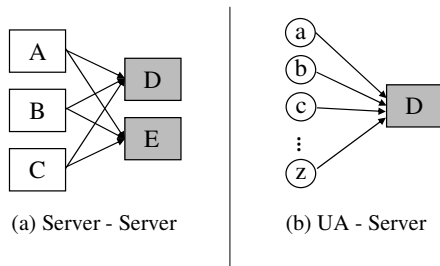


Fig. 2. SIP server topologies.

exponential back off to compensate for lost messages. SIP differentiates between two types of transactions: INVITE and non-INVITE transactions. The two transaction types differ in the way requests are retransmitted and confirmed. Each time a SIP entity sends a request over an unreliable transport protocol to the next hop, it starts a retransmission timer with a default value of 500ms. This timer is called timer *A* for INVITE and timer *E* for non-INVITE transactions. When this timer expires, the request is retransmitted and the timer is reset with its value doubled. For non-INVITE transactions, the value of timer *E* is capped and is not raised above a given maximum (the default is 4s). Retransmissions cease when a response is received or the request timeout timer fires, which has a default value of 32s. The request timeout timer is called timer *B* for INVITE and timer *F* for non-INVITE transactions and is used independent of the transport type. For INVITE transactions, the UAS retransmits the 200 OK response until it receives an ACK using the capped exponential back off mechanism as in timer *A* and a request timeout as in timer *B*. 200 OK responses are retransmitted independent of the transport protocol in use. With these retransmission schemes, a SIP message can be transmitted up to seven times without a cap for timer *A/E* and eleven times when the doubling of timer *A/E* is capped.

B. SIP Server Topologies

The topology of SIP servers impacts overload control in SIP. Figure 2 (a) depicts a server-server configuration in which a SIP server *D* receives traffic from multiple upstream neighbors. Each of the upstream servers forwards potentially many requests to the downstream server. In this topology, an overload control mechanism can notify each upstream server to reduce the load forwarded. An upstream server can have an alternative downstream server it can forward a request to if the first server is unavailable. The UA-server scenario is depicted in Figure 2 (b). Here, a SIP server receives requests from a large population of UAs. Each user agent typically only generates a small number of requests, which are often related to the same call. Hence, asking user agents to lower the number of requests has little impact on the load of server *D*. However, since INVITE transactions consist of multiple requests, rejecting an INVITE can still lower the offered load of server *D*. In this paper, we focus on server-server overload control and use local overload control in our simulations for UA-server configurations if needed.

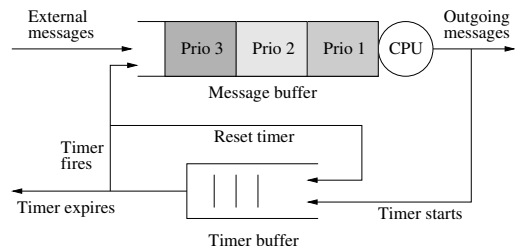


Fig. 3. Queuing model for a stateful server.

IV. SIMULATION MODEL

We have implemented a discrete-event simulator to evaluate the behavior of SIP servers under overload. Our simulator implements the full SIP transaction layer and key elements of a transaction user for INVITE and non-INVITE transactions according to the SIP RFC 3261 [16]. This simulation model provides us the flexibility to evaluate the impact of different SIP server topologies, server processing capacities, overload control algorithms and internal message processing techniques such as the stateless rejection of SIP requests. Our simulator has been carefully evaluated and verified in the design team for SIP overload control, which was formed by the IETF SIPPING working group. We also verified the results of our simulator for Network Topology 1 (see below) with the goodput of a SIP proxy implementation, the SIP Express Router (SER) [19], in the same configuration.

A. Model for SIP Servers

When a SIP entity sends new request, it instantiates a client transaction, which is governed by a finite-state machine (FSM). Similarly, when a SIP entity receives a request, it instantiates a server transaction that is again governed by a finite-state machine. In addition to the client or server transaction, a SIP entity also instantiates an INVITE or non-INVITE transaction user. Four types of FSMs are defined depending on whether the transaction is at the client or server, and whether the transaction is triggered by an INVITE or non-INVITE message. A description of these four FSMs can be found in Section 17 of [16]. The SIP servers are set to operate in transaction stateful mode. UAs and SIP servers can transmit messages either via UDP or TCP. If UDP is used, retransmission timers are used as defined by SIP. Servers are set to record route, which causes all SIP messages exchanged between two UAs to traverse through these servers. This includes the ACK request and the BYE transaction, which would otherwise be exchanged directly between UAs. Since the initial INVITE request and responses and their retransmissions always traverse SIP servers, the use of record-route does not qualitatively change the results.

The structure of the queuing system for a SIP server is shown in Figure 3. External messages (i.e., requests and responses) and timer messages (contexts) are queued in a message buffer, which implements a non-preemptive priority queuing discipline with priority i having a higher precedence over priority j if $i < j$. Messages of the same priority

are queued in a FIFO manner. The processor (CPU) serves the message at the head of the priority queue by executing the necessary FSM, sending a message to the next hop, and possibly starting a timer. Timers are placed in a priority queue sorted according to their firing times. When a timer fires and thus exits the timer buffer, its associated context is queued into the message buffer if required by the FSM, and a new timer with a subsequent firing time is enqueued in the timer buffer if needed. A timer that expires simply leaves the system.

Generally, timer messages are assigned priority 1 while external messages are assigned priority 2. Priority 3 is needed for local overload control and is discussed in Section VI. Because some timer messages need to be processed to ensure that the associated transactions are properly cleaned, we assume that the space for timer messages is unlimited. External messages can be queued up to B messages, and new external messages are dropped if its space is full.

We assume the following parameters for the “base” server. The external message service time at the processor is 1ms and the timer message service time is 0.5ms. This implies that the retransmission of a SIP message takes half the time of the initial transmission. A server with a given processing capacity can be defined in the simulator by scaling up/down the values given in the base server. We assume that message service time is deterministic. Our experiment with exponential message service time does not show qualitative differences. Note that while packet service time in a router typically depends on payload length and not on header processing, SIP message service time is mainly determined by header processing rather than payload length.

All associated timer firing intervals use the default values specified in RFC 3261. Throughout this paper, we assume that $B = 1000$, and if high watermark B_h and low watermark B_l are used, the values are 800 and 600, respectively.

B. Model for UAs

Our model uses an infinite number of UAs and each new INVITE request is generated by a new UA instance. INVITE requests are created with an aggregate rate of λ requests/second according to a Poisson process. Because other messages are generated based on events (e.g., timer firing or receiving a message) and the state of the FSM, their arrival processes is governed by the dynamics of interacting SIP transactions. If a call is not established 10 seconds after the INVITE request was sent it is assumed that a user will abandon the call and the INVITE transaction is cancelled. The holding time for an established call is assumed to be exponentially distributed with parameter $1/E[T_h]$, where $E[T_h]$ is the average holding time. We assume $E[T_h] = 100$ secs. Finally, a UA sends a BYE request to terminate its session. The offered load is the total number of calls (new INVITEs) per second initiated by UAs. The goodput is the total number of calls per second terminated by UAs.

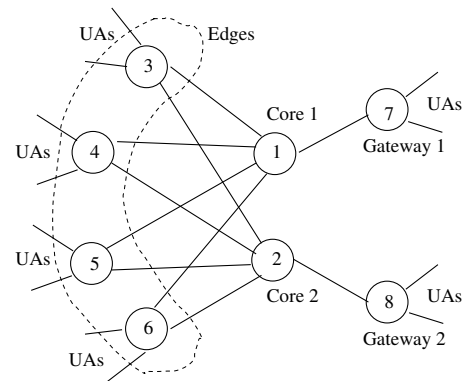


Fig. 4. Network topology 2.

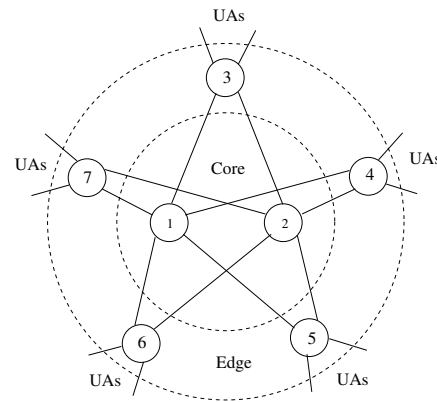


Fig. 5. Network topology 3.

C. Network Topologies

The simplest topology (Network Topology 1) we have used in our experiments is a single server serving all the UAs; that is, each pair of UAs are communicating via a single server. This simple star topology is useful to study the behavior of a SIP server in isolation.

The next network topology is depicted in Figure 4. It consists of M edge servers ($M = 4$ for example) serving UAs that are interested in communicating with users served by one of two gateway servers. Each edge server has to select one of the core servers to reach the respective gateway server. This configuration can be used, for example, to connect SIP UAs to PSTN gateways.

Figure 5 shows a topology with M edge servers, which send traffic through one of N core servers. The edge servers balance load across the core servers. Edge servers forward traffic from the UAs to the core servers and vice versa. Each UA is connected to one edge server. Edge servers can exchange messages with all core servers. This topology is typically used for UAs that communicate within a SIP service provider domain.

An extension of the preceding topology is one where core servers in one domain forward messages to core servers in another domain, as shown in Figure 6. The above network topologies are representations of topologies proposed in stan-

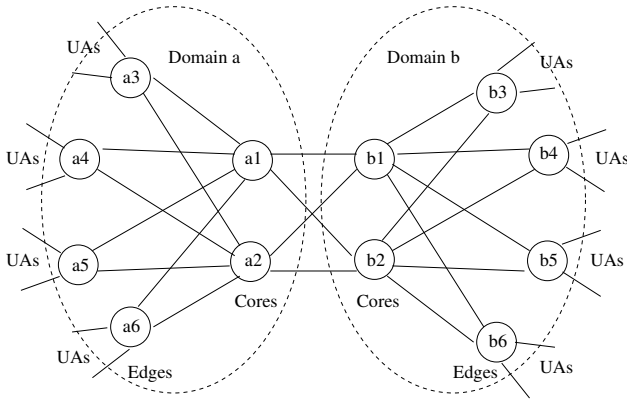


Fig. 6. Network topology 4.

dards, e.g., in the IMS architecture [1] and are used by SIP service providers.

SIP servers communicate via an underlying IP network. The IP network may drop packets and introduce packet delay due to queuing. We do not consider the effects of the underlying network in order to isolate the performance behavior due to SIP. We also assume that propagation delay in the network is negligible.

V. SIP PERFORMANCE EVALUATION

In this section, we explore the performance of SIP-based networks without additional overload control mechanisms.

A. Congestion Collapse and Recovery from Congestion Collapse

We first begin with the simplest topology, Network Topology 1, in which all UAs are communicating via a single server. We assume that this server simply discards arriving messages when its buffer is full. With the parameters set to those of the base server described in the previous section and the call flow depicted in Figure 1, the server has a capacity of about 160 calls per second (cps).

Figure 7(a) shows a sample path of the goodput as a function of time when the offered load to the server is varied. After a delay that is due to the call holding time the goodput initially tracks the offered load at about 150 cps. As the offered load exceeds the server capacity at $t = 1000$ s, the system collapses and the goodput drops significantly. At this point, the server starts to drop messages which are in turn retransmitted by the sender. These retransmissions amplify the offered load, which eventually leads to a congestion collapse. Interestingly, when the offered load is reduced back to 150 cps (i.e., below the server capacity) at $t = 2000$ s, the congestion collapse persists and the goodput remains low.

Figure 7(b) depicts the timer firing rate over time. Although the rate of new call arrivals has been reduced after $t = 2000$ s, internal as well as external message arrivals due to timer retransmissions remain high and contribute to load that keeps the server under stress. (Since timer B and F fire at a much lower rate than timer A and E, their arrival rate remains at the

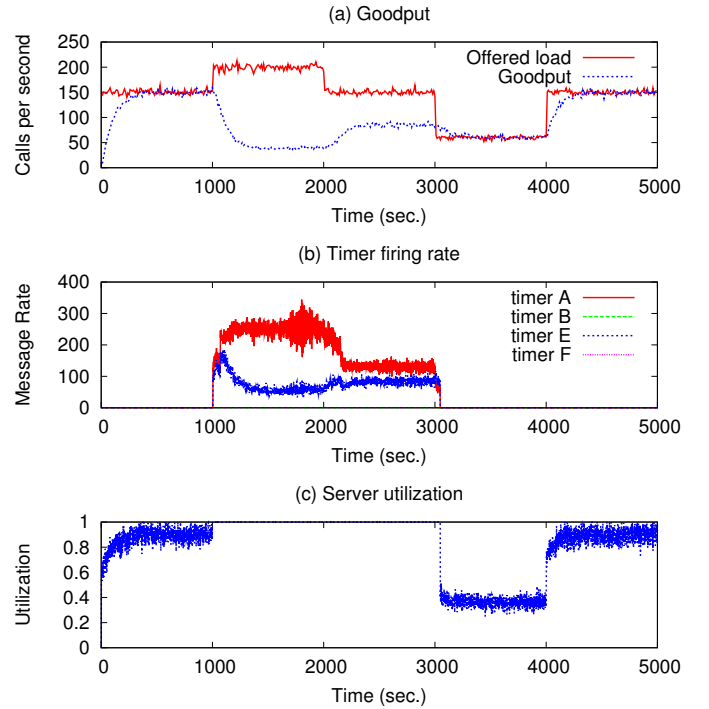


Fig. 7. SIP server performance over time as the offered is varied: (a) the call goodput drops when the offered load exceeds the capacity of the server and does not recover easily, as (b) message retransmissions due to timer firing exacerbate the overload and (c) result in very high server utilization.

bottom of Figure 7(b).) This is also corroborated in Figure 7(c) where the server utilization remains close to 1 even after reduction of the call arrival rate. Only after the offered load is reduced sufficiently at $t = 3000$ s the timer retransmissions are cleared. At this point, the server recovers from congestion collapse.

B. Cascading Effects

We now investigate how a congestion collapse at a particular server in a network affects other servers. We use Network Topology 2 to illustrate the interactions among different servers in a network. We assume that all edge servers and core server 1 each have a capacity of 80 cps, while the gateways and core server 2 each have a capacity of 160 cps. Traffic is sent from UAs connected to the gateways to the UAs connected to the edge servers and vice versa. All traffic is uniformly distributed among the four edge servers. The traffic between the edge servers and gateway 1 is routed through core server 1. Traffic between the edge servers and gateway 2 is routed through core server 2. Denote the overall offered load exchanged between the edge servers and the gateway i by λ_i ($i = 1, 2$).

Figure 8(a) plots the offered loads and the corresponding goodputs for both traffic streams. Initially at $t = 0$, $\lambda_1 = 65$ cps and $\lambda_2 = 80$ cps. Calls are successfully set up as both core servers have ample capacities to process the traffic. Note in Figure 8(c) that core server 1 already operates at a high utilization while the other servers are moderately loaded.

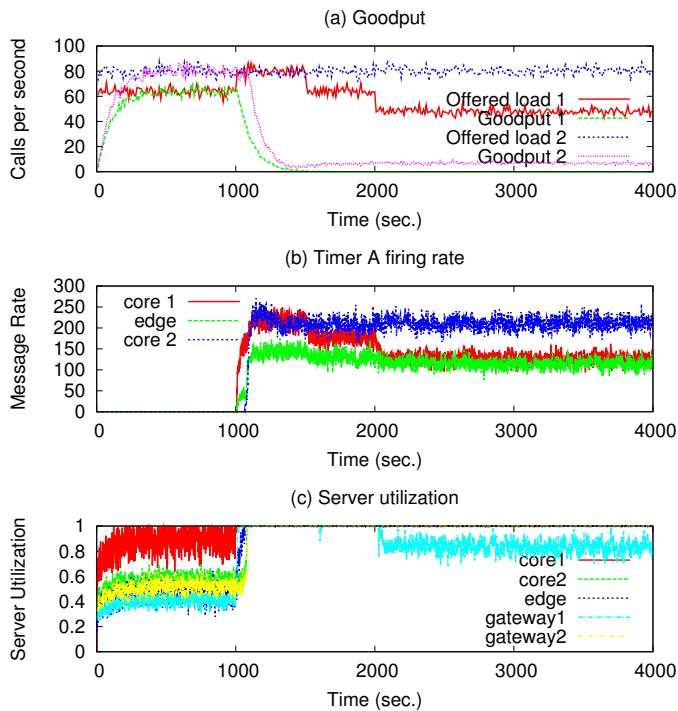


Fig. 8. Effect of one SIP server on other servers: (a) when offered load 1 exceeds the capacity of a server, goodput 2 for another path can be impacted, (b) confirmed by message retransmissions at different servers, and (c) utilizations at different servers.

At time $t = 1000$ s, λ_1 is increased slightly above 80 cps, which causes core server 1 to be overloaded. λ_2 remains at 80 cps. Because core server 1 is experiencing a congestion collapse, each of the edge servers will start to retransmit many of its messages, as can be seen in Figure 8(b). This leads to a congestion collapse at the edge servers. Subsequently, as the edge servers become unresponsive, core server 2 starts to perform retransmissions for requests and responses sent and eventually goes into congestion collapse. As can be seen after $t = 2000$ s, even after the offered load λ_1 is reduced, the servers remain in a state of congestion collapse.

C. Controlling Overload with the 503 Response Code

The use of the 503 response code with a Retry-After header generates an on/off traffic pattern since a SIP server alternates between not forwarding requests and forwarding all requests. The on/off pattern of 503 responses with Retry-After leads to instability in the offered load and can be expected not to offer a satisfactory performance. During periods in which traffic cannot be forwarded, a SIP server can reject requests with a 500 (Server Internal Error) response or retry them at an alternate server. Retrying at alternate servers can cause traffic oscillation among the servers.

To demonstrate the effects of using the 503 response code with Retry-After for overload control, we use Network Topology 3. We assume that a server receiving a 503 response from a downstream neighbor will retry the request at an alternate server, if possible. Calls are rejected with a 500 (Server

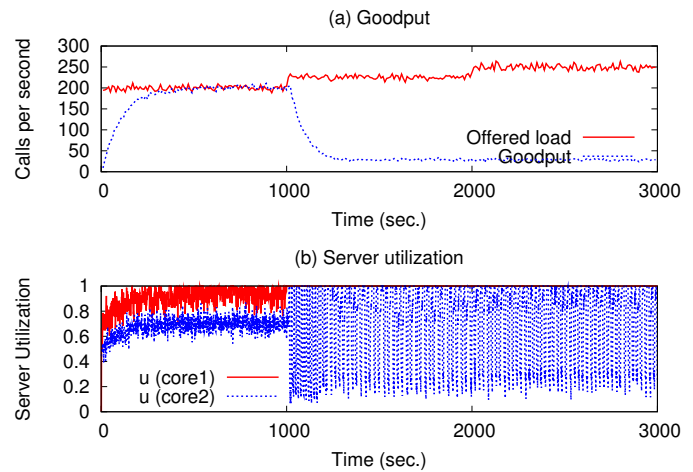


Fig. 9. Use of 503 responses to control overload: (a) goodput drops significantly during overload, and (b) servers may oscillate.

Internal Error) response code if they are unsuccessful at all downstream servers. We set the capacity of core server 1 to 120 cps and core server 2 to 160 cps. Each of the edge servers has a capacity of 480 cps to ensure that only core servers are overloaded. Edge servers are set to balance load across the two core servers. A server starts to send 503 responses to its upstream neighbor if the server's queue length has reached the high-watermark level. A server stops sending 503 responses after the queue length has decreased below the low-watermark level. Figure 9(a) plots the total goodput and offered load using Retry-After value of 10s. We use 10s since it allows us to better examine the traffic oscillations that occur. The observed effects are, however, the same even with the shortest possible value of 1s. As can be seen, this technique results in congestion collapse during overload. Figure 9(b) shows the utilization at each core server. It reveals that core server 2 is oscillating at a period of about 10s. Core server 1, which is slower, is experiencing a severe overload and is unable to recover from the overload condition during the Retry-After timeout periods. It still needs to process responses as well as internal messages during these times. Oscillation is generally a bad behavior as it leads to poor performance and provides inconsistent services to the end users. We investigate the use 503 responses without a Retry-After header in the context of local overload control.

D. Impact of Transport Protocols on SIP Performance

Since SIP can use a reliable (e.g., TCP) or unreliable (e.g., UDP) transport protocol, it is of interest to evaluate SIP performance under different transport protocols and, in particular, TCP versus UDP. Many of the SIP retransmission timers are only activated if SIP is used over an unreliable transport protocol. Therefore, we expect that the goodput of a SIP server under overload will be higher if TCP is used instead of UDP. To confirm our expectation, we compare three scenarios. In the first scenario, we assume that all SIP entities along the path of a request are using TCP. In the second scenario, we exploit the fact that the SIP protocol allows

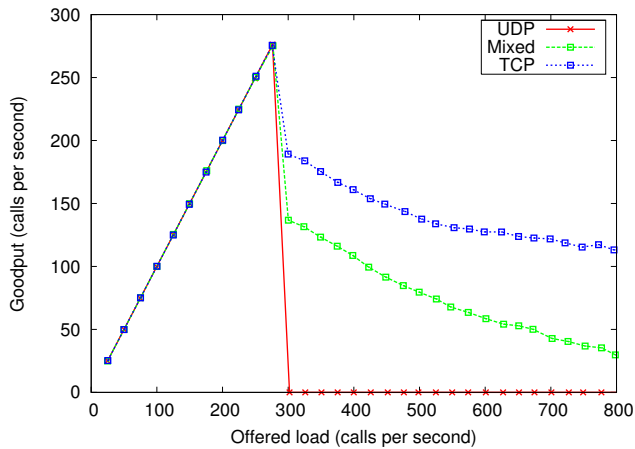


Fig. 10. Goodput comparison for different transport protocols.

the use of a different transport protocol on each hop. Here, UAs use UDP to reach a SIP server while the server-to-server communication uses TCP (this scenario is labeled the *mixed* model). In the third scenario, we only use UDP.

Figure 10 illustrates the goodput versus offered load for the three different scenarios using Network Topology 3. It is assumed that each server has a capacity of 160 cps. Traffic is uniformly distributed across the core servers and edge servers. As expected, the goodput performance progresses as more and more TCP links are used. Notice that even with TCP, we still see a significant drop in goodput once the server capacity is reached. This can be attributed in part to the fact that a UAS retransmits 200 OK responses with an exponential back off irrespective of the transport in use. Another reason is related to the way TCP is used. During overload, a SIP server can either keep reading from a TCP socket and discard some of the SIP messages it receives or it can stop reading SIP messages from the TCP socket altogether. We have used the former method as it decouples network congestion from overload control. It enables the SIP server to continue to process incoming messages such as responses, ACK and BYE requests and only discard new INVITE requests. This method is used in many SIP proxy implementations. The latter method implicitly uses TCP as a feedback mechanism for overload control that affects all messages from a server. We leave this method for further study.

While there are many considerations that determine the choice of a transport protocol, our studies confirm that using a reliable transport provides a better performance under overload for a SIP server.

VI. LOCAL OVERLOAD CONTROL

A mechanism that is frequently used to improve performance of a SIP server under overload is local overload control.

A. Overview

The basic idea of local overload control is that a SIP server, which is getting close to its capacity limit, starts to reject SIP requests locally with 503 responses without Retry-After

header instead of dropping messages or using 503 responses with Retry-After. The 503 response code with no Retry-After header only affects the rejected request. Unlike the two other methods, rejecting requests and sending 503 responses upstream consumes processing resources. Nevertheless, rejecting a request can be made to consume far less resources than fully processing it. Even if rejecting requests consumes less processing resources, an overloaded SIP server with local overload control will eventually spend most of its processing resources on rejecting requests if the offered load is very high, which leads to a poor goodput.

The rationale behind local overload control is that it is beneficial to avoid that upstream servers retransmit copies of messages that have been dropped and thereby amplify the offered load. Instead of dropping these messages, local overload control suppresses retransmissions by rejecting them. Furthermore, since a server can reject each request individually, it can freely choose the fraction of requests it wants to reject. This enables a smooth control of the requests to be processed. Local overload control does not require cooperation between servers. This characteristic is important since it enables the use of local overload control even if upstream servers are non-cooperative.

To speed up the rejection process for local overload control, we do not use a transaction-layer FSM in SIP servers when rejecting these requests with 503 responses without Retry-After. In other words, a SIP server rejects requests for local overload control in a stateless mode, which avoids that state needs to be looked up or created.

We noted that the stateless generation of 503 responses can create a race condition between a 200 success and 503 failure response for the same request, which eventually causes the affected call to fail. To further investigate the impact of these call failures on the overall performance, we also performed simulation runs using the standard stateful generation of 503 responses. We found that stateless 503 response generation can provide a marginal performance improvement but there was no qualitative difference in the performance of the two methods.

Another important aspect of the rejection process is whether requests are selected for rejection due to overload “early”, i.e., before enqueueing them in the input message buffer or “late”, i.e., when they are dequeued. To explore the first alternative, we have configured our queuing model described in Section IV-A by assigning priority 1 for timers, priority 2 for requests selected for rejection and priority 3 for all other messages. Thus, only requests that are admitted by local overload control are enqueueing in the input message buffer; all other requests are scheduled to be rejected. For “late” rejections we use our standard queuing model with 2 priorities. We found that a combination of early rejections and stateless rejections provides the best performance given all other parameters remain the same and we use those two message processing techniques in our evaluation described below.

B. Algorithms for Local Overload Control

A key component of local overload control is the algorithm that determines the amount of requests a SIP server should accept. We compare the use of two algorithms for local overload control (other possible algorithms are described, for example, in [9]). The first algorithm is a simple *bang-bang control (BBC)* algorithm that is also described in [11]. Here, a server has two states: (1) overload and (2) underload. A server in underload state turns to overload state when its message queue length exceeds a high watermark value B_h . Similarly, a server in overload state turns to underload state when the queue length falls below a low watermark value B_l . In underload state, all messages are accepted whereas in overload state, all incoming INVITE requests are rejected.

The second algorithm is called the *occupancy (OCC)* algorithm [3], where incoming INVITE requests are accepted with probability f (or rejected with probability $1 - f$). The OCC algorithm is based on processor occupancy (utilization) ρ . It has the objective of dynamically adjusting f to maintain a utilization at or below a given target utilization ρ_{target} . The utilization, ρ , is periodically updated at every τ seconds. In each t -th epoch, the processor utilization ρ_t is updated and compared with the target utilization ρ_{target} . If desired, the utilization can be smoothed with a moving average filter. The basic idea of OCC is to increase f if $\rho < \rho_{target}$, and to decrease it otherwise. Let f_{t+1} denote the newly updated f in the next epoch $t + 1$, while f_t denote f in the current epoch t . The algorithm that updates f in each epoch is described as follows.

$$f_{t+1} = \begin{cases} f_{min}, & \text{if } \phi_t f_t < f_{min} \\ 1, & \text{if } \phi_t f_t > 1 \\ \phi_t f_t, & \text{otherwise,} \end{cases} \quad (1)$$

where f_{min} represents the threshold for the minimum fraction of traffic accepted. The multiplicative increase/decrease factor ϕ_t is given by

$$\phi_t = \min\{\rho_{target}/\rho_t, \phi_{max}\}, \quad (2)$$

where ϕ_{max} defines the maximum possible multiplicative increase in f from one epoch to the next.

C. Performance Comparison

In this section, we compare four different cases:

- Case 1: The base case where no overload control is employed.
- Case 2: Local overload control with BBC.
- Case 3: Same as Case 2, except that an upstream server will re-try a rejected request at an alternate server, if possible.
- Case 4: Local overload control with OCC.

We use Network Topology 3 and set the capacity of each server to 160 cps. Each call to an ingress edge server is uniformly distributed to an egress edge server. Edge servers distribute calls equally across the core servers. We assume that an early reject incurs a processing time of 0.16666ms. For the OCC algorithm, $f_{min} = 0.02$, $\phi_{max} = 5$, $\rho_{target} = 0.9$, and ρ_t is updated every second.

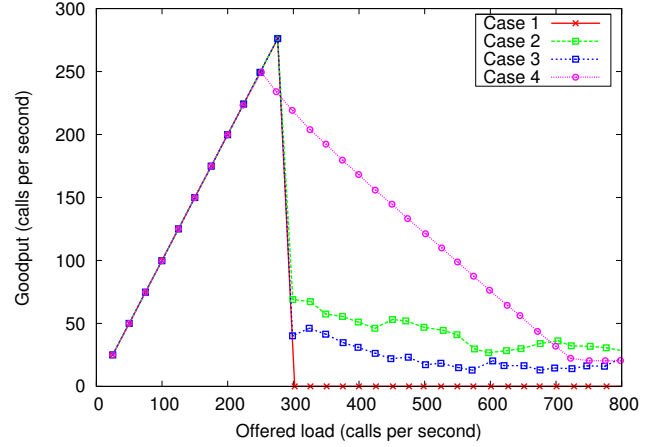


Fig. 11. Goodput comparison based on topology 3. Case 1: no overload control; Case 2: BBC without alternate server; Case 3: BBC with alternate server; Case 4: OCC without alternate server.

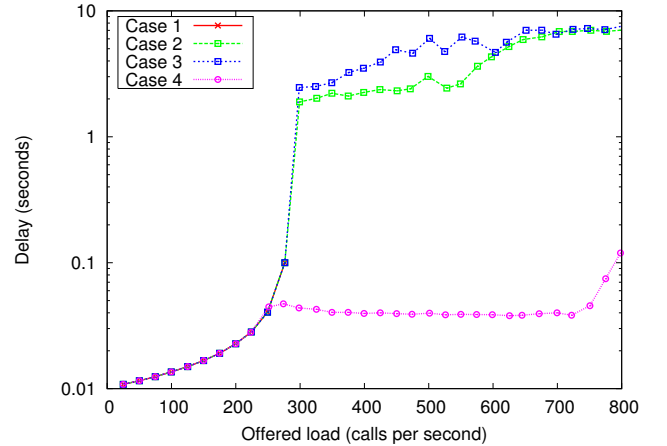


Fig. 12. Delay comparison based on topology 3. Case 1: no overload control; Case 2: BBC without alternate server; Case 3: BBC with alternate server; Case 4: OCC without alternate server.

As can be seen in Figure 11, the goodput with Case 1 increases up to the network capacity and then rapidly drops to zero with a severe congestion collapse at around 300 cps. The corresponding mean setup delay (the difference between the time an INVITE is sent by a UAC and the time an ACK is received by a UAS) is depicted in Figure 12. Note that for Case 1 the delay is plotted only up to about 300 cps as no calls can get through beyond it.

Case 2 in Figure 11 depicts the goodput provided by the BBC algorithm without re-trying rejected requests at an alternate server. The goodput reaches close to the server's processing capacity at first since in BBC a server does not begin to reject requests until the buffer has reached the high-watermark and the server turns into overload state, which is near its capacity. Once a server enters overload state, it starts to reject all INVITE requests, resulting in a rather sudden drop in goodput. Another reason for the overall poor performance of BBC is that the processor is allowed to reach a very high

utilization. With local overload control the processor needs to have enough capacity to reject requests, which may not be available in BBC at the time the buffer occupancy is at the high watermark. It is also interesting to note that the corresponding setup delay in Figure 12 increases near the capacity limit as the utilization approaches 1, but levels off due to finite buffering.

In the previous cases, edge servers were programmed to reject all requests for which they have received a 503 response from a core server with a 500 response. In Case 3, edge servers will retry all rejected requests at the second core server. Only if a request is rejected there as well, a 500 response will be returned back to the UAC. This behavior is suggested in the SIP specification.

The graphs labeled Case 3 in Figure 11 and Figure 12 plot the goodput and delay if re-routing is allowed. Contrary to the initial intuition, retrying decreases goodput and increases delay during overload here. This can be explained by the fact that retrying rejected requests at alternate servers increases the load on these servers. Each server receives all requests that were rejected by the other server. The more alternate servers are available, the worse is this effect since a request is sent to all servers before it is cleared from the system.

Retrying requests can be beneficial when one SIP server is highly loaded while the alternate server has plenty of capacity. However, in many cases the performance of such a configuration can be improved by adjusting the distribution of load across servers instead of retrying requests during overload.

Case 4 in Figure 11 plots the goodput of the OCC algorithm. Note that the goodput degrades approximately linearly as the load increases. This is because OCC increasingly rejects INVITE requests as the offered load increases. The OCC algorithm does not fully reach the capacity limit of the server since it starts to reduce the number of requests processed when ρ reaches 0.9. The delay shown in Figure 12 indicates that OCC is able to maintain a low delay for the calls that are processed as long as the load stays below the rejection rate of the server. Thus, while an increasing number of calls are rejected, the ones that are processed do not see a significant increase in setup delay. However, when the load increases beyond the rejection capacity, the delay starts to increase significantly. This can be explained by the fact that the server starts to drop requests at this point and enters into a congestion collapse.

Interestingly, the goodput of OCC, as well as of all other control algorithms, does not drop to zero when the offered load reaches the rejection capacity of a server. This can be attributed to the Poisson distribution of call arrivals. Even under a high load, there are periods which the offered load is low enough to allow the server to process a few incoming INVITE requests.

Summing up, we find that local overload control is capable of extending the range under which a server can operate without dropping requests. In particular, if an appropriate control algorithm is used, local overload control outperforms a system without overload control and enables a server to cope with light cases of overload. However, local overload control

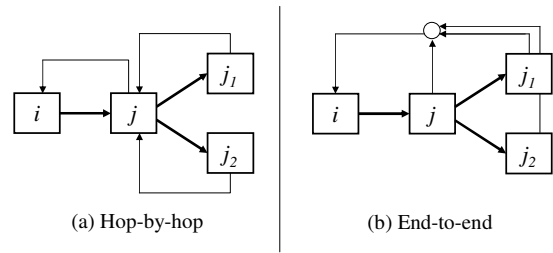


Fig. 13. Hop-by-hop vs. end-to-end overload control.

does not prevent a congestion collapse and therefore is not suitable as the sole overload control mechanism.

VII. DISTRIBUTED OVERLOAD CONTROL

Distributed overload control allows an overloaded server to offload the task of rejecting requests to other servers.

A. Hop-by-hop Vs. End-to-End

In distributed overload control, SIP servers are enabled to provide overload control feedback to servers that are further upstream in a SIP server network. This feedback can be used by the upstream servers to reduce load to an amount that does not cause downstream servers to be overloaded. Overload control feedback can, for example, be conveyed in a SIP response header [8]. The overload control feedback loop can be applied to the path of a SIP request hop-by-hop, i.e., individually between each pair of SIP servers, or end-to-end as a single control loop that stretches across the entire path from UAC to UAS. Figure 13 depicts the two alternatives.

In the hop-by-hop mechanism, a separate overload control loop is instantiated between each pair of neighboring SIP servers on the path of a SIP request. For example in Figure 13(a) a separate overload control loop is established between servers j_1 - j , j_2 - j , and j - i . Each SIP server provides feedback to its direct upstream neighbors, which then adjust the amount of traffic they are forwarding to this SIP server. The upstream neighbors do not forward the received information upstream. An upstream neighbor uses a separate overload control loop with its own upstream neighbors. In this model, overload is always resolved by the direct upstream neighbors of an overloaded server without the need to involve entities that are located multiple SIP hops away.

In our experiment, each SIP server j independently reports its current acceptance probability f^j value to its upstream neighbors in SIP responses¹. An upstream neighbor of j simply accepts new INVITES that are to be forwarded to j using the reported probability f^j (or rejects them with probability $1 - f^j$)². If an upstream neighbor i is at the network edge, it will instead use $\min\{f^i, f^j\}$, where f^i is its own computed value. Note that if the set of direct downstream neighbors of a server is (j_1, j_2, \dots, j_L) , the server only has to

¹In general, a server can report a rate cap instead of acceptance probability or even raw load information such as utilization and queue length.

²In practice, a server needs to be able to handle upstream neighbors that do not support overload control. This is not addressed in this paper.

maintain a list of $(f^{j_1}, f^{j_2}, \dots, f^{j_L})$ values. Thus, the hop-by-hop mechanism scales well to networks with many SIP entities.

The end-to-end overload control mechanism on the other hand implements an overload control loop along the path of a SIP request. When a SIP server receives the first request from its upstream neighbor, it knows the downstream neighbor to forward the message to, but not the entire path toward the destination. Our end-to-end mechanism makes use of this fact. As there are different sets of downstream neighbors for different target servers (those that are one-hop away from the UASs), each server needs to keep track of its downstream neighbor's information on a per-target basis. Specifically, for a given target server d , a server j maintains a list of $(f^{j_1(d)}, f^{j_2(d)}, \dots, f^{j_L(d)})$ values if its corresponding downstream neighbors for target d are $(j_1(d), j_2(d), \dots, j_L(d))$, where $f^{j_i(d)}$ is the f value the server receives from its downstream neighbor j_i for target d . If server j with the preceding downstream neighbors reports the feedback information to its upstream neighbor, it will only report the summarized information $f^{i(d)} = \min\{f^{j_1(d)}, f^{j_2(d)}, \dots, f^{j_L(d)}, f^i\}$. The feedback information for each target is eventually propagated to all ingress servers. An ingress server i makes the decision on accepting new INVITES for target d based on the smaller of f^i value or the value received from its downstream neighbor. The end-to-end mechanism is in particular suitable for networks with fixed call routing policies. The main disadvantage of the end-to-end mechanism is its greater complexity compared to hop-by-hop mechanism.

B. Performance Comparison

We use Network Topology 4 to compare local, hop-by-hop and end-to-end overload-control mechanisms. For each case, the OCC algorithm is used. We assume that each server has a capacity of 160 cps. The offered load among all edge servers is uniformly distributed with routing to each core server equally likely.

Figure 14 compares the goodput performance for the different overload-control mechanisms. As expected, hop-by-hop significantly outperforms local overload-control. Nevertheless, the hop-by-hop mechanism still experiences some loss of goodput as the offered becomes very high. On the other hand, the end-to-end mechanism maintains high goodput throughout. It is intuitively clear that end-to-end mechanism will not offer a performance advantage over the hop-by-hop mechanism when an overloaded server is only one-hop away from an ingress server, as in Network Topology 3. At the other extreme, the performance of the hop-by-hop mechanism can be expected to be lower if an overloaded server occurs in the middle of a long path whereas the end-to-end mechanism maintains high goodput as long as the ingress servers are not overloaded.

VIII. CONCLUSION

In this paper, we have investigated the performance of a network of SIP servers under overload. The motivation behind our work is that, the servers using the current SIP

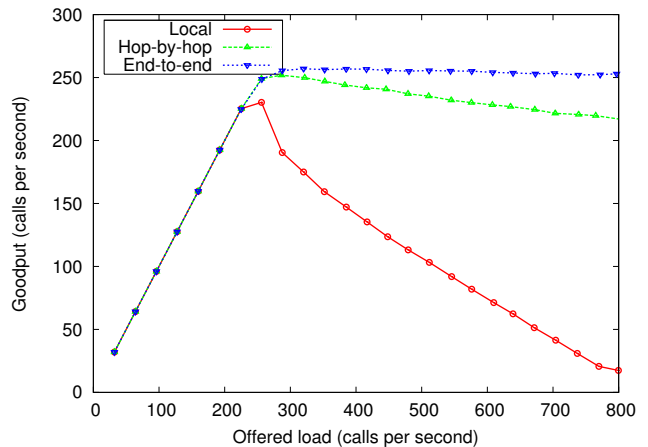


Fig. 14. Comparisons among different overload-control mechanisms.

protocol are vulnerable to overload and congestion collapse. This shortcoming is now becoming apparent as SIP networks are deployed in different domains on a large scale. To the best of our knowledge, our work provides the first comprehensive study of SIP overload which reveals problems such as spreading of overload and slow recovery of SIP servers. Our work also provides a side-by-side comparison of different overload control algorithms (BBC without alternate servers, BBC with alternate servers, OCC) and overload control paradigms (local, hop-by-hop and end-to-end overload control).

We have shown that the current overload control mechanism of SIP is unable to prevent congestion collapse and may, in fact, worsen an overload condition. Due to message retransmissions triggered by various SIP timers, overload may spread throughout a network of SIP servers and SIP servers cannot easily recover from overload once it has occurred. Our results have shown that using local overload control techniques can provide a simple remedy for light cases of overload; however, it is ineffective to treat higher amounts of load. Finally, we have investigated distributed overload control mechanisms, which are able to prevent congestion collapse in a network of SIP servers under a wide range of overload. Based on our results we argue that an extension of the SIP protocol for overload management is necessary and feasible.

REFERENCES

- [1] 3rd Generation Partnership Project, <http://www.3gpp.org>.
- [2] M. Allman, V. Paxson and W. Stevens, "TCP Congestion Control," IETF, RFC 2581, April 1999.
- [3] B. L. Cyr, J. S. Kaufman and P. T. Lee, "Load balancing and overload control in a distributed processing telecommunication systems," United States Patent No. 4,974,256, 1990.
- [4] B. T. Doshi and H. Heffes, "Overload performance of several processor queueing disciplines for the $M/M/1$ queue," *IEEE Transactions on Communications*, vol. 34, no. 6, pp. 538-546, Jun. 1986.
- [5] R. P. Ejzak, C. K. Florkey and R. W. Hemmeter, "Network overload and congestion: a comparison of ISUP and SIP," *Bell Labs Technical Journal*, vol. 9, no. 3, pp. 173-182, 2004.
- [6] ETSI, "Architecture for Control of Processing Overload," ETSI, DTR/TISPAN-02026-NGN, 2006.
- [7] S. Floyd, T. Henderson and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," IETF, RFC 3782, April 2004.

- [8] V. Hilt, I. Widjaja and H. Schulzrinne, "Session Initiation Protocol (SIP) Overload Control," IETF, Internet-Draft, draft-hilt-sipping-overload, 2008.
- [9] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari and T. LaPorta, "Fast and robust signaling overload control," *International Conference on Network Protocols*, Riverside, CA, November 2001.
- [10] D. R. Manfield, G. K. Millsted and M. Zukerman, "Performance analysis of SS7 congestion controls under sustained overload," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 3, pp. 405-414, Apr. 1994.
- [11] M. Ohta, "Overload Control in a SIP Signaling Network," *Enformatika Transactions in Engineering, Computing and Technology*, Mar. 2006.
- [12] R. R. Pillai, "A distributed overload control algorithm for delay-bounded call setup," *IEEE Transactions on Networking*, vol. 9, no. 6, pp. 780-789, Dec. 2001.
- [13] K. Ramakrishnan, S. Floyd and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," IETF, RFC 3168, September 2001.
- [14] I. Rhee and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variants," *PFLDnet*, Lyon, France, 2005.
- [15] J. Rosenberg, "Requirements for Management of Overload in the Session Initiation Protocol," IETF, Internet-Draft, draft-ietf-sipping-overload-reqs, 2008.
- [16] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, "SIP: Session Initiation Protocol," IETF, RFC 3261, June 2002.
- [17] M. Rumsewicz, "On the efficacy of using the transferred-controlled procedure during periods of STP processor overload in SS7 networks," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 3, pp. 415-423, Apr. 1994.
- [18] M. Schwartz, "Telecommunication networks," Addison-Wesley, Reading, MA, 1987.
- [19] SIP Express Router, <http://www.iptel.org/ser/>
- [20] D. Tow, "Network Management-Recent Advances and Future Trends," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 4, May 1988.
- [21] M. Welsh and D. Culler, "Adaptive Overload Control for Busy Internet Servers," *USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.
- [22] W. Zhao and H. Schulzrinne, "Enabling On-demand Query Result Caching in DotSlash for Handling Web Hotspots Effectively," *IEEE Workshop on Hot Topics in Web Systems and Technologies*, Boston, Massachusetts, November 2006.