Novel Overload Controls for SIP Networks

Eric Noel
AT&T Labs Research
Middletown, NJ 07748
Email: eric.noel@att.com

Carolyn R Johnson
AT&T Labs Research
Middletown, NJ 07748
Email: carolyn.johnson@att.com

***Abstract*—The increasing use of SIP in Next Generation Networks necessitates that SIP networks provide adequate control mechanisms to optimize transaction throughput and prevent congestion collapse during traffic overloads. SIP elements need well behaved internal overload controls to detect and throttle traffic, and external controls between SIP elements are needed to throttle traffic closer to the source. This paper presents the results of simulations that demonstrate the inadequacy of current SIP controls, and then presents several novel algorithms to control SIP network overloads. We show our new algorithms increase goodput across increasing load levels, are robust to IP loss and delay, and perform well under transient burst loads.**

I. Introduction

Session Initiation Protocol (SIP) is a message based signaling protocol that has been widely adopted for use in Internet based applications. SIP is used for session oriented applications, including Voice over IP (VoIP), multimedia sessions, video conferencing, presence, and instant messaging. SIP was standardized by the IETF [1], and related RFCs. The 3GPP standards body has adopted SIP as the signaling protocol for its Internet Multimedia Subsystem (IMS) architecture [2].

The growth of internet telephony is increasing significantly as service providers more widely deploy services over IP to share infrastructure. The increasing use of SIP in Next Generation Networks requires that SIP based networks provide adequate control mechanisms for handling traffic growth. In particular, SIP networks must be able to handle traffic overloads gracefully, optimizing transaction throughput without causing congestion collapse. Recent work in the IETF [3] has identified significant inadequacies in the current SIP ability to handle overloads. SIP has a basic overload control mechanism that allows on-off throttling of traffic. However, our results described in this paper and recent publications [3], [4], [5] demonstrate the current SIP control mechanism is ineffective in managing network overloads. Work in the IETF continues to address the need for effective overload control mechanisms so that standards based solutions can be developed and deployed.

As SIP has grown in popularity, end users and traffic has increased on SIP based networks. As with other communications networks, SIP networks are typically designed and engineered to handle the traffic load peaks from busy hour traffic. The ability to handle some level of additional load due to network element failures and/or traffic peaks is also a key part of well engineered networks. For example, with commonly used n+1 or more general n+k sparing designs of network elements, communications networks are designed to handle the loss of an element without degrading the network throughput. However, it is not cost effective to engineer large scale networks for extreme traffic surges. Sudden increases in traffic volumes can be caused by various events, including mass calling triggered by natural disasters or emergencies, and media stimulated events such as voting for contestants during a limited period of time. In addition, SIP end points simultaneously registering to SIP servers due to power outages or failures could cause overload. To handle the overload cases, SIP solutions need effective overload controls. Two types of controls are needed. First, network elements must have internal overload controls as a mechanism to detect and throttle traffic when the traffic exceeds the system capacity. Internal overload controls alone are insufficient when traffic loads are excessive. Even with well behaved internal overload controls, congestion collapse can result due to message retransmissions from the sources. Therefore, external overload controls between network elements are needed to throttle the traffic as close to the source as possible. In this paper, we address both internal and external overload controls, but the focus is to demonstrate effective external overload controls to mitigate congestion.

As described in our earlier work [6], overload controls have been widely studied and deployed in traditional transaction based networks. Typical methods of overload control include basic throttling such as percent blocking, and more sophisticated methods such as Automatic Code Gapping (ACG) or window based controls. These controls have been finely tuned in traditional networks to optimize network efficiency when the network experiences overloads. We will use traditional controls as a basis for improving the SIP protocol to better handle overloads. As mentioned, SIP has a limited capability to control network overloads with an on-off type of control. When a SIP element has reached its maximum capacity and cannot process requests, it responds to new session requests with a 503 Retry After message. This instructs the source to stop sending messages for the current session, but does not stop the source from sending subsequent session requests. SIP provides an optional 503 Retry After timer parameter to stop the source from sending new session requests for a specified period. However, as demonstrated in this paper, the on-off 503 Retry After method is unable to adequately address overloads.

In [6] we showed the throughput benefit for queue based controls. Ohta [7] showed similar improvement with internal queue controls, but did not include external SIP feedback controls. In [4], queue based on-off and occupancy based

external controls show similar improvements. Our work includes additional algorithms, and is used to support the IETF [3] design work for SIP controls. Additionally, this paper provides new results that quantify impacts of underlying IP impairments, as well as non-uniform traffic distributions and traffic bursts.

In this paper, we present the results of simulation models that demonstrate congestion collapse with current SIP capabilities. Then we focus on several novel algorithms and methods for managing overloads in SIP networks. We consider both methods that minimize the changes needed to the current SIP RFC as well as new methods that would require enhancing the SIP RFC. The remainder of this paper will present these results. First, the SIP message flow, SIP network elements, and network topology are described. Then results of the current SIP 503 Retry After control and our new control algorithms are presented. Finally, sensitivity analyses on impacts of the underlying IP transmission impairments, traffic distribution of impacts, and transient behavior are presented.

II. Modeling SIP Protocol For Overload Control

SIP is an IETF standardized signaling protocol for creating, modifying, and terminating sessions between SIP entities [1]. It consists of three layers: a Transaction User that initiates and responds to requests, a Transaction Layer where the SIP message exchange rules reside, and a Transport Layer that encapsulates messages in an underlying transport protocol (UDP, TCP or SCTP).

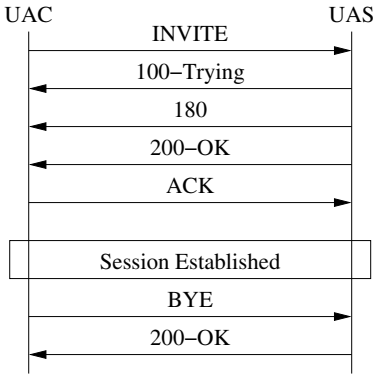

Fig. 1. INVITE-BYE call flow between a User Agent Client/Server pair (intermediate proxies not shown for conciseness).

Being concerned with overload in SIP telephony networks, we consider SIP dialogs for establishing and tearing down voice calls between User Agents (UAs) across one or more SIP proxies. Following SIP RFC 3261 [1], we implemented in a simulation model the state machines that manage SIP dialogs associated with the standard INVITE-BYE message flow (see Figure 1).

Because SIP is often deployed over unreliable transport, we only consider SIP over UDP, so handling of error corrections via retransmissions takes place in both the Transaction Layer and the Transaction User layers (as opposed to Transport Layer when reliable transport is used).

SIP uses timers for application-level retransmissions and message timeouts. Retransmission timers are only used with UDP and rely on a short timer initialized to T1 (0.5sec) that is doubled after each retransmission.[1] A long timer proportional to T1 ($64 \times$ T1sec) is used for message timeouts.

The SIP RFC provides the 503 response method for overload control. When in overload, a proxy will send a 503 response to reject new call requests (initial INVITE requests) while messages for existing calls are preserved. Upon receiving a 503 response, a proxy will stop sending requests to the overloaded server for a number of seconds defined in the Retry-After header (if any), retry an alternative proxy if one is available, or reject the request back to the UAC.[2]

We report results for the stateless 503 mode of operation[3] which is the method for rejecting new call requests with the least impact in capacity. The penalty incurred is the possibility for state inconsistencies among proxies. This issue was resolved in [8] by the introduction of a new state in the INVITE server Transaction Layer state machine.

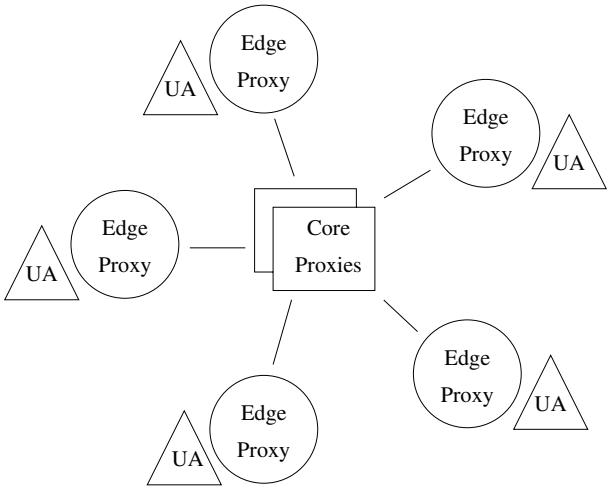

Fig. 2. Network benchmark [9].

To evaluate our proposed overload control algorithms, we used the same benchmark network as the one developed by the IETF sip-overload design team [9] and represented in Figure 2. In this network of 5 Edge Proxies and 2 Core Proxies, calls originate and terminate between UAs and traverse a pair of Edge Proxies and a Core Proxy. Edge Proxies and UAs are paired so that calls from or to an UA will traverse the same Edge Proxies. Selection of Core Proxies is random. All proxies are assumed to be stateful, so signaling messages within the same call traverse the same set of proxies.

In order to focus this work on controlling overloads in the Core Proxies, both UAs and Edge Proxies are assumed to be of infinite capacity. Overload controls to throttle UAs are not addressed here. Clearly, overload controls to protect Edge Proxies when subject to congestion induced by UAs (each

[1]Short timer is capped by timer T2 = 4sec for non-INVITE messages.
[2]Calls originate from a UA Client and terminate to a UA Server.
[3]No "memory" of sending a 503 response is kept.

account for a few call attempts) would be very different than those associated in protecting Core Proxies, our main focus.

We model Core Proxies by a queueing system where all signaling messages enter the same queue (of size 500 messages) and are serviced by a single server in First-In-First-Out order. We used the same internal overload control algorithm as that defined in [9], namely, as soon as the input queue reaches a high watermark (set to 400 messages) all new call requests are rejected until the input queue drains to a low watermark (set to 300 messages). To reflect the reduced processing of rejected INVITEs, the message service rate for rejected INVITEs is 3,000 msg/sec and 500 msg/sec for accepted INVITEs and all other messages. Message service rate and queue length were selected so that the maximum queueing delay (1 sec) exceeds SIP retransmission timer T1, while message service rate for rejected INVITEs was arbitrarily selected to be significantly smaller than that of accepted INVITEs.

So each Core Proxy maximum capacity corresponds to 500 msg/sec divided by 7 messages per call, or a total of ≈ 143cps.

Results reported next were derived from our simulator[4] that implemented the model aspects presented previously. In the results sections, each data point corresponds to a simulation run of $\approx 3 \times 10^6$ call attempts with the reported statistics reset after the first 0.5×10^6 call attempts to cancel simulation initialization effects. Comparing simulation results from multiple independent runs allowed use to estimate the error due to our simulations inherent randomness to be $< 5\%$ of the reported statistics.

Lastly, call attempts were Poisson distributed with exponential holding time of mean 3minutes (typical for voice applications) and uniformly distributed across all User Agents.

III. Overload Control Algorithms For SIP

A. Preliminaries

In Section II, we described an internal overload control algorithm which basically consists of a mechanism used by an overloaded proxy to reduce offered load with no support from any other element. This is in contrast with an external overload control (loosely referred to as overload control throughout this article) where an overloaded proxy would rely on an external throttle point to reduce offered load.

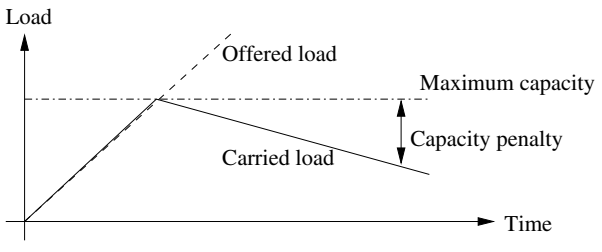

Fig. 3. Illustration of the penalty on call processing capacity incurred from rejecting calls in a proxy.

[4]Discrete event simulator with a heap sort scheduler written in C of about 15,000 lines.

The idea of external throttle point stems from the notion that an overloaded element should dedicate all its capacity in processing calls and that the capacity penalty from rejecting calls should be delegated to external throttle points (see Figure 3). Moreover, we expect a "well behaved" overload control to produce a goodput[5] as close as possible to the theoretical maximum, independently of overloading level. Clearly, a proxy could be designed so that the penalty for rejecting new calls is negligible, although in practice such designs are rare.

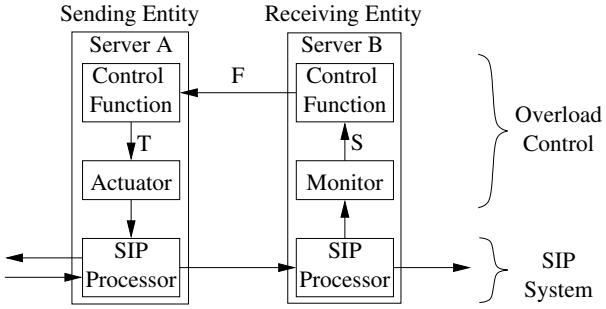

Fig. 4. System model for overload control [3].

Throughout this article, we use the system model in Figure 4 (taken from [3]) to describe our overload controls. There, the monitor function gathers internal measurements for the overload control algorithm. The control function, based on the monitor function input, determines the overload state, estimates target offered load or how much the offered load must be reduced, and sets the actuator parameters to achieve the desired goal. While the actuator is responsible for rejecting offered load towards the receiving entity per the control function specifications.

All our overload controls rely on a feedback loop from the overloaded element to the throttle point (flow F between servers B and A in Figure 4). That way, the throttle point can dynamically adjust the load destined to the overloaded element to allow for operation at nearly optimal capacity.

In general, we set the overload control so that it is activated prior the internal overload control.

B. SIP RFC Overload Control

With the SIP RFC overload control, both monitor and control functions are located in the Core Proxies while the actuators are located in the Edge Proxies. The monitor tracks the Core Proxy input queue length, while the control function compares the queue length to the high watermark queue level (400 messages). Upon identification of the high watermark threshold crossing, the control function declares the Core Proxy in overload, randomly draws a retry-after timer between 0 and 10sec (as specified in the SIP RFC) and forwards it to the actuators in 503 response messages. The Core Proxy will remain in overload until its queue length becomes less than the low watermark threshold (300 messages).

[5]Following [9], defined as the rate of successfully processed calls such that 200-OK is received by originating Edge Proxies within 10sec of sending the initial INVITE (see Figure 1).

When the retry-after timer is in effect, actuators reject all new call requests towards the overloaded Core Proxies.

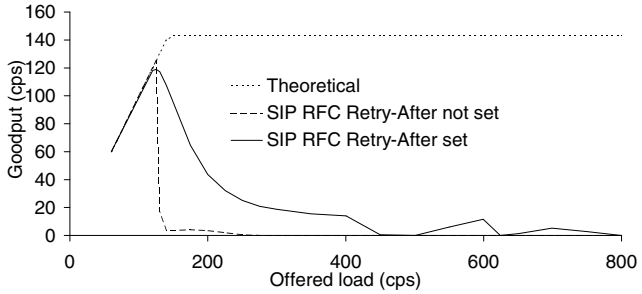

Fig. 5. Comparison between SIP RFC overload control goodput and the maximum goodput with increasing offered load.

In Figure 5, we compare the maximum allowed goodput to the one produced by the SIP RFC overload control with or without optional 503 retry-after timer set by Core Proxies and honored by Edge Proxies. These results confirm the congestion collapse phenomenon reported in [3] [4] and [5].

Note that congestion collapse could be postponed by tuning the internal overload control watermarks such that the high watermark queueing delay is less than that of the SIP retransmission timer initial value of T1. However, increasing the offered load will eventually result in congestion collapse.

C. Retry-After Control

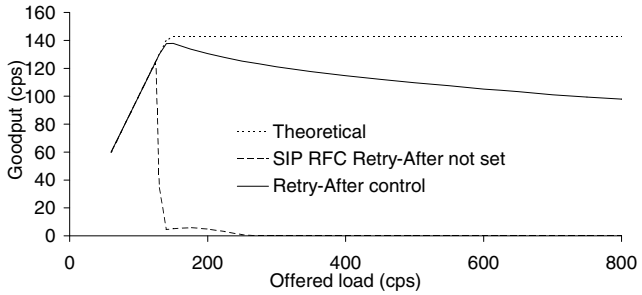

Fig. 6. Comparison between our retry-after control goodput and the maximum goodput with increasing offered load.

Here, our goal was to define an overload control algorithm that requires as little as possible modifications from the SIP RFC overload control. So we attempted to improve setting the retry-after timer from that defined in the SIP RFC to a value that depends on the overloaded proxy load. We found that setting the retry-after timer to the duration needed for the overloaded proxy input queue to drain its current level to a low watermark level achieved significant goodput improvements, as depicted in Figure 6.

Our retry-after control differs from the SIP RFC overload control as follows. In the monitor, each sampling interval t (of duration $T = 0.1$sec), the mean service message rate

$$\mu_t = \sum(\text{Processed messages in } t)/T$$

and the queueing delay

$$d_t = (\text{Queue length at end of } t)/\mu_t$$

are measured. If d_t exceeds the high watermark αd_e, the control function declares the Core Proxy to be in overload and rejects all new call requests with a 503 message until the queueing delay becomes less than the low watermark βd_e.

μ_t	Mean service rate in msg/sec
α, β	High and low watermark parameters set to 0.9 and 0.1 respectively
T	Sampling period duration set to 0.1sec
d_e	Target queueing delay set to 0.1sec

TABLE I
SUMMARY OF PARAMETERS USED IN THE RETRY-AFTER CONTROL.

The queueing delay threshold d_e was set to 0.1sec, deliberately smaller than the internal overload high watermark (400 messages or ≈ 0.5sec). The watermark parameters α and β were set to 0.9 and 0.1 respectively. We summarize all the parameters used in our retry-after control in Table I.

The control function sets the retry-after timer to the maximum between $d_t - \beta d_e$ and $(\alpha - \beta)d_e$ and embeds it in 503 responses passed to Edge Proxies actuators.

So when Edge Proxies reject new call requests destined to the overloaded Core Proxies for the retry-after duration, it cannot receive any 503 response messages back from the overloaded Core Proxies until expiration of the retry-after interval. Hence a "time lag", independent of offered load, between when the Core Proxies get out of the overload state and when the Edge Proxies stop throttling. That "time lag" is responsible for the goodput degradation as function of increasing overload level noticeable in Figure 6.

Note we could have achieved goodput improvements by having the Core Proxies notify the Edge Proxies of its overload state changes using a dedicated control message (i.e. SIP NOTIFY) or response messages from calls processed before the retry-after period (if any). However, the goal of the control being to minimize the impacts on the SIP standard, we did not make these improvements.

D. Processor Occupancy Control

Next we propose a rate control based on processor occupancy. There, the overloaded element calculates a target call rate based on its processor occupancy and broadcasts the rate to each throttle point within any SIP response message.

As with the previous controls, both monitors and control functions are located in the Core Proxies, while the actuators are located in the Edge Proxies. Each sampling interval t, the monitor measures the mean service message rate μ_t exactly the same way as in the retry-after control. It also measures the processor occupancy

$$U_t = (1-w)U_{t-1} + w(\text{Processor busy time during } T)/T$$

and the mean number of messages per call

$$r_t = (1-w)r_{t-1} + w(\text{Number of messages received in } t)/(\text{Number of new call requests in } t)$$

where $w = 0.8$. During the same sampling interval, the control function compares U_t to the high watermark αU_e, where U_e is the target processor occupancy. If that watermark is exceeded, the control function declares the Core Proxy to be in overload until the measured processor occupancy becomes less than the low watermark βU_e.

μ_t	Mean service rate in msg/sec
U_t	Processor occupancy
r_t	Mean messages per calls
A_t	Mean number of active sources
N_t	Number of new call attempts
λ_t	Target call rate
α, β	High and low watermark parameters set to 0.9 and 0.1 respectively
w	Moving average parameter set to 0.8
T	Sampling period duration set to 0.1sec
U_e	Target processor occupancy set to 0.9

TABLE II
SUMMARY OF PARAMETERS USED IN THE PROCESSOR OCCUPANCY CONTROL.

We set the target processor occupancy U_e to 90%, so at most 90% of the Core Proxy maximum capacity can be achieved.[6] In Table II, we summarize all parameters used in our processor occupancy control.

The control function sets the target message rate to $U_e\mu_t$ and the target call rate λ_t to $U_e\mu_t/r_t$.

Before forwarding a target rate to the actuators, the control function must scale the target call rate by the number of active sources. So, we estimate[7] the number of active sources as

$$A_t = (1-w)A_{t-1} + wA_{t-1}N_{t-1}/(T\lambda_{t-1})$$

where N_{t-1} is the number of new call attempts in $t-1$ and the expression $T\lambda_{t-1}/A_{t-1}$ corresponds to the expected number of calls per active sources in $t-1$.

Several candidate algorithms exist as means of throttling traffic in the actuators. The most popular ones are call gapping, percent blocking, leaky bucket and window control ([10] [11] [12]), each of various implementation complexity and inherent fairness (i.e. with percent blocking all traffic sources are subject to the same blocking probability, while with leaky bucket blocking is proportional to the amount of load in excess of the target load). Since each can "track" a target rate sufficiently well and we will not consider fairness in this paper, we selected a percent blocking throttling algorithm for it simplicity to implement.

So before sending a call request to an overloaded Core Proxy, the actuators apply the percent blocking algorithm with parameter p_b estimated as

$$p_b = (\lambda_{\text{offered}}/\lambda_t) - 1$$

where λ_{offered} is the call rate incoming to the Edge Proxy destined towards the overloaded Core Proxy and measured over a sampling period of duration T.[8]

If the result is to block the call request, the actuator notifies the UAC of the call rejection.

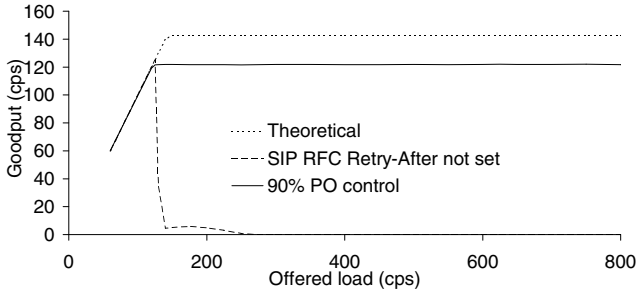

Fig. 7. Comparison between our processor occupancy control goodput and the maximum goodput with increasing offered load.

In Figure 7, we compare our processor occupancy control goodput to the theoretical maximum for 90% target occupancy. We found that, unlike with the retry-after control, the processor occupancy control goodput remains at 90% of the theoretical maximum independently of the overloading load level.

E. Queue Delay Control

Our queue delay control is also a rate control that operates the same way as the processor occupancy control. But instead of relying on processor occupancy to estimate target call rate and identify overload state, it uses queue delay.

The queue delay control monitor functions measure the service rate μ_t and the queueing delay d_t the same way as in the retry-after control. It also measures the mean message per call r_t the same way as in the processor occupancy control. While the control function relies on the same queue delay watermarks (αd_e and βd_e) used in the retry-after control. The control function also estimates the target call rate as

$$\lambda_t = (\mu_t/r_t)(1 - (d_t - d_e)/T).$$

Our expression for target call rate can be interpreted as the Core Proxy capacity discounted by the capacity required to drain the queue below the target queueing delay.

μ_t	Mean service rate in msg/sec
r_t	Mean messages per calls
A_t	Mean number of active sources
N_t	Number of new call attempts
λ_t	Target call rate
α, β	High and low watermark parameters set to 0.9 and 0.1 respectively
w	Moving average parameter set to 0.8
T	Sampling period duration set to 0.1sec
d_e	Target queueing delay set to 0.1sec

TABLE III
SUMMARY OF PARAMETERS USED IN THE QUEUE DELAY CONTROL.

We summarize all parameters used by the queue delay control in Table III.

[6]Note that a processor occupancy target too large would produce an unstable queueing system.

[7]An alternate method could be to count the number of active sources and weigh the target rate by the proportion of traffic from each traffic source.

[8]Note, Core Proxies and Edge Proxies sampling periods are not synchronized.

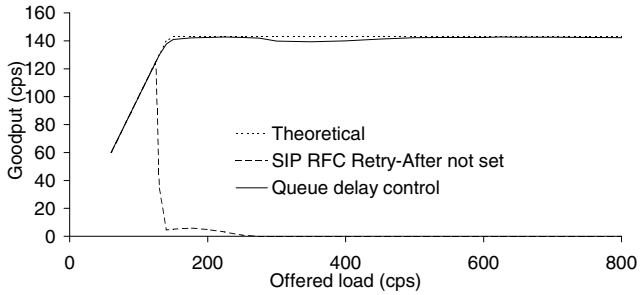

Fig. 8. Comparison between our queue delay occupancy control goodput and the maximum goodput with increasing offered load.

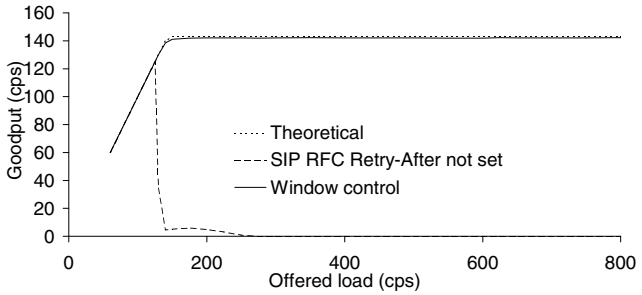

Fig. 9. Comparison between our window control goodput and the maximum goodput with increasing offered load.

Comparing our queue delay control goodput to the theoretical maximum in Figure 8, we found our rate control to track nearly perfectly the maximum theoretical goodput.

F. Window Control

Unlike the previous controls, all window overload control functions are in the Edge Proxies. So only the internal overload control resides in the Core Proxies.

The actuator forwards a new call to a Core Proxy if and only if the number of outstanding call requests W_o towards that Core Proxy is strictly less than the corresponding window size W_s. Otherwise the actuator notifies the UAC of a call rejection.

W_s	Window size
$W_{\max}$, $W_{\min}$	Window bounds set to 100 and 0.5 respectively
W_+	Increment for W_s, set to 0.1
W_-	Decrement for W_s, set to 0.5
C	Counter of positively acknowledged call requests
$C_{\max}$	Upper bound for C, set to 2
W_o	Number of outstanding call requests
Q_h, Q_l	Core Proxies high and low queue length watermarks set to 100 and 50 respectively

TABLE IV
SUMMARY OF PARAMETERS USED IN THE WINDOW CONTROL.

The control function uses a counter mechanism to increment or decrement W_s that is strongly biased to favor reducing W_s. Any new call request positively acknowledged (with a 100-Trying) results in incrementing C and any call request timed out or negatively acknowledged (with a 503) results in setting C to 0 and decrementing W_s by W_-. Once C reaches $C_{\max}$, W_s is incremented by W_+ and C is set to 0.

Because we allow for real numbered window sizes, it is possible to have zero outstanding call requests and a non-zero window size less than 1. When such condition occurs, we emulate "fractional" calls by allowing an additional call with probability the window size.

Simulation results showed that to maximize the window control goodput, Core Proxy internal overload control must be set so it activates before its input queue delay is "too close" to T1, so we heuristically set the high and low queue length watermarks for internal overload control to 100 and 50 respectively (as opposed to the original settings of 400 and 300). See Table IV for a summary of the parameters used in our window control.

Comparing our window control goodput to the theoretical maximum in Figure 9 shows that our window control tracks very close to the maximum theoretical goodput.

IV. SENSITIVITY ANALYSIS

A. Transmission Impairments

Here, we quantify the impacts of message loss and transmission delay on our overload control algorithms. Independent of our controls, the SIP protocol provides a retransmission algorithm to account for delayed or lost INVITEs, 200-OKs and BYEs. So both message loss and transmission delay beyond retransmission timeout produce an increase in the number of messages per call that results in a decrease in call rate capacity.

So we expect an overall degradation of goodput as a function of increasing delay or message loss. To validate our results we estimated upper bounds for the maximum goodput for our network benchmark as a function of either message loss or transmission delay and plotted the results in Figure 10 (curves labeled theoretical). To estimate our upper bounds, we assumed Core Proxies input queue to be empty and ignored 503 processing penalties.

Inspection of simulation results for loss in Figure 10 show that our controls are impacted similarly. At 114cps, all controls but the processor occupancy control produce well grouped goodput curves close to the maximum goodput and larger than that of the processor occupancy. While at 1,000cps, both window and queue delay controls produce nearly identical goodput control that exceed retry-after and processor occupancy controls goodput. For the delay curves in Figure 10, we find the retry-after control to be most sensitive to transmission delay and to produce a goodput significantly less than that of the other controls with increasing delay.

B. Traffic Distribution

	Call Completion Rate (Case 3 Overloading Sources)			
	Retry-After	90% PO	Queue delay	Window
Network wide average	9.3%	12.2%	14.2%	14.2%
Overloading source	8.8%	8.3%	10.5%	12.6%
Engineered source	19.2%	93.8%	90.8%	47.6%

TABLE V
CALL COMPLETION STATISTICS FOR NON UNIFORMLY DISTRIBUTED TRAFFIC SOURCES.

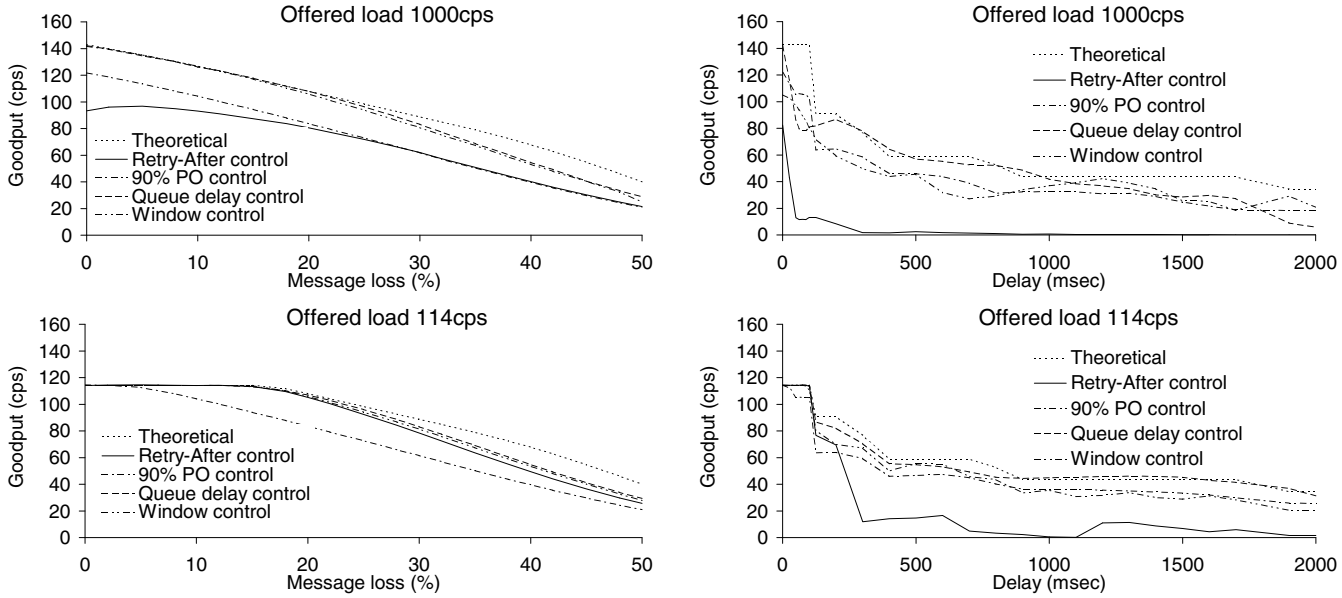

Fig. 10. Transmission impairment sensitivity of overload controls (left most column: Message loss, right most column: Delay).

Next, we look at the effect of non uniformly distributed traffic sources. To do so, we define a subset of the sources (UAs grouped with five Edge Proxies) as engineered sources that produce a combined volume of 114cps (80% of the two Core Proxies maximum capacity). And the rest of the sources as overloading sources so that their combined volume plus that of the engineered sources is 1,000cps.

Simulation results showed that the engineered sources call completion rates[9] for the processor occupancy and the queue delay controls were $\approx$ 90% while that of the overloading sources was significantly degraded (Table V). This can be attributed to the rate based nature of the processor occupancy and queue delay controls, where the throttle points limit the offered load towards the Core Proxies to the target call rate. Both the window and retry-after controls also yield better call completion rate for the engineered source, but not as significant as with the rate controls.

Note that by incorporating a fairness policy, we could set our overload controls to produce identical call completion rate across all sources, to penalize either source type or to maximize call completion rate of a specific source. However, this is not covered in this article.

C. Transients

Lastly, we compare our overload controls transient behavior. To do so, each control is subject to a step offered load that begins at 114cps (80% of maximum capacity) for 5min, followed by a 1000cps step for 5min and a decrease to 114cps for another 5min. In Figure 11, we show single simulation run transient goodput curves with a sampling period of 5sec.

There, the retry-after control activates gracefully, but due to its inherent lag limitation, can never produce an overload goodput at maximum capacity. The queue delay and window controls also activate gracefully and maximize overload goodput. While the processor occupancy control suffers from a significant goodput loss prior activation then achieves an overload goodput at 90% of maximum capacity.

Careful observation shows that goodput significantly exceeds the maximum capacity for about 3 minutes. This is due to the lag between call setup messages and call release messages that lasts approximately the holding time duration of mean 3 minutes. So at the beginning of the overload, Core Proxies are subject to the superposition of a steady state load of 7 messages per calls at 114cps and a burst of 5 messages per call at 886cps, that is an equivalent maximum capacity of ≈ 155cps.

	Response Time (msec)		Call Completion (%)	
Control	Activate	Deactivate	Theoretical	Actual
Retry-after	194.1 ± 1.1	228.4 ± 3.4	30.2	25.6
90% PO	610.8 ± 1.8	$5,399.9 \pm 72.8$	29.1	28.8
Queue delay	204.9 ± 1.3	351.2 ± 1.1	30.2	30.8
Window	278.2 ± 0.7	24.4 ± 0.8	30.2	30.7

TABLE VI
TRANSIENT STATISTICS. (RESPONSE TIMES MEASURED AT THE EDGE PROXIES, CALL COMPLETION CALCULATED OVER THE ENTIRE SIMULATION RUN.)

In Table VI, we quantified the transient behavior of our overload controls. It turns out the processor occupancy takes most time to activate and deactivate while the other three controls take about the same amount of time to activate. The window control deactivates the fastest.

V. CONCLUSIONS

This paper presents the results of analysis on SIP networks under overload and several new overload controls. The key features of the SIP message call flow, SIP elements, and SIP network topology are defined. Each of the SIP elements is assumed to have well behaved internal overload controls to detect and throttle excess traffic when its capacity is exceeded. These internal overload controls use high and low watermarks

[9]We define call completion rate as the number of good calls divided by the number of call attempts over a simulation run.

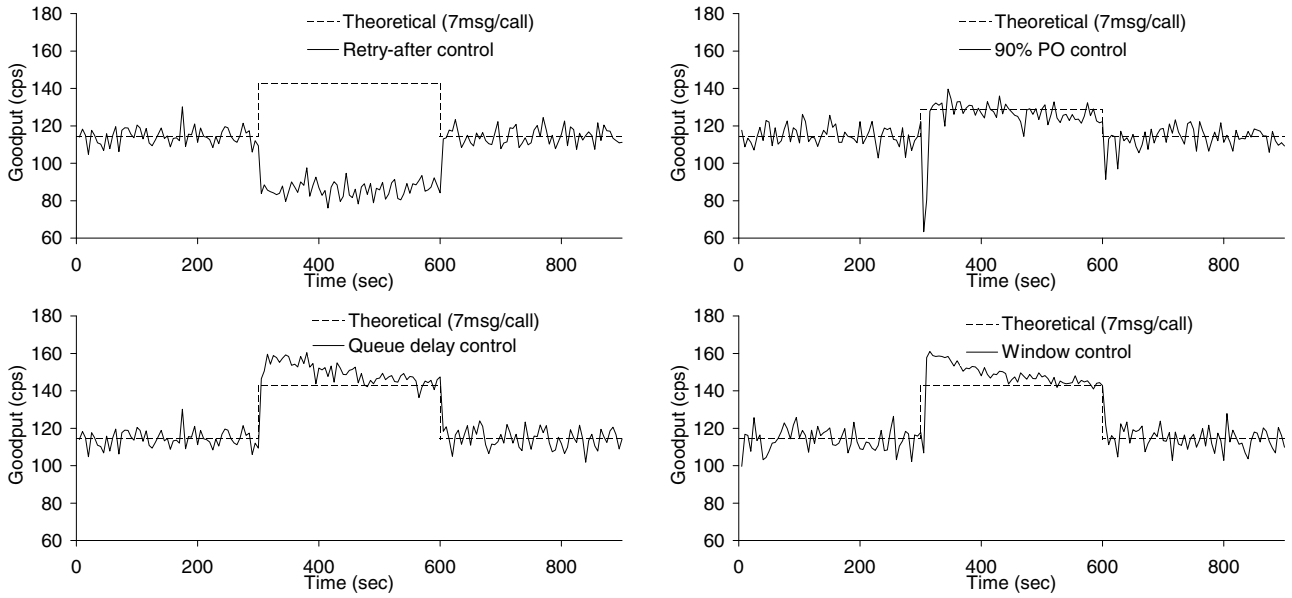

Fig. 11. Transients for overload control when subject to a step offered load varying between 114cps and 1000cps (sampling period 5sec).

as parameters to start and stop throttling traffic. External overload controls are needed to throttle traffic as close to the source as possible. The current SIP 503 Retry After external control method is analyzed to demonstrate its inability to prevent congestion collapse. Next Retry After Control is proposed with significant goodput improvement, as a method of improving performance with minimal impact to the SIP protocol. However, the time lag inherent in this on-off control method results in sub-optimal goodput performance. Next, new controls that introduce new messages from the overloaded element toward the sources are presented. Our processor occupancy control improves goodput because it reduces the effect of time lag. Our queue delay and window based controls demonstrate near optimal goodput performance.

Finally, we investigated the impacts of several factors important in real SIP over IP networks. IP networks experience packet loss and delay, which impacts SIP message flows. We characterized goodput for each of the controls as a function of message loss and delay, and as a function of uniform and focused source overload. Then we evaluated overload transient behavior for each of the controls using goodput as the metric. In general, the Retry After control experienced increased sensitivity to delay and transient load, while queue delay and window controls experienced the best goodput performance.

In future publications, we will address sensitivity on network size, element queue sizes and processing rates. We will also include other metrics such as call setup delays, message drop rates, queue length statistics and processor utilization.

ACKNOWLEDGMENT

The authors would like to acknowledge the IETF sip-overload design team that we collaborated with to derive the benchmark network used to evaluate our proposed overload control algorithms.

REFERENCES

[1] 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.

[2] "3rd Generation Partnership Project". http://www.3gpp.org.

[3] V. Hilt Editor. "Design Considerations for Session Initiated Protocol (SIP) Overload Control". *IETF,* draft-ietf-sipping-overload-design-01, March 2009.

[4] V. Hilt, I. Widjaja. "Controlling Overload in Networks of SIP Servers". *IEEE International Conference on Network Protocols (ICNP)*, October 2008.

[5] C. Shen, H. Schulzrinne, E. Nahum. "SIP Server Overload Control: Design and Evaluation". *Principles, Systems and Applications of IP Telecommunications (IPTComm)*, July 2008.

[6] E. Noel, C.R. Johnson. "Initial simulation results that analyze SIP based VoIP networks under overload." *ITC*, (2007) 5464.

[7] M. Ohta. "Overoad Control in a SIP Signaling Network". *Enformatika Trans. On Engineering, Computing and Technology*, vol. 12, pages 205–210, 2006.

[8] R. Sparks. "Correct transaction handling for 200 responses to Session Initiation Protocol INVITE requests". *IETF*, draft-sparks-sip-invfix-02, July 2008.

[9] J. Rosenberg & V. Hilt coordinators. "IETF sip-overload Design Team".

[10] A. Berger. "Comparison of Call Gapping and Percent Blocking for Overload Control in Distributed Systems and Telecommunications Networks". *IEEE Transaction on Communications*, vol. 39, No 4, pages 574–580, April 1991.

[11] A. Berger. "Overload Control Using Rate Control Throttle: Selecting Token Bank Capacity for Robustness to Arrival Rates". *IEEE Transaction on Automatic Control*, vol. 36, No 2, pages 216–219, February 1991.

[12] A. Hác, L. Gao. "Analysis of congestion control mechanisms in an intelligent network". *Int. Journal of Network Management*, vol. 8, No 1, pages 18–41, 1999.