Application Layer Feedback-based SIP Server Overload Control

Charles Shen
Department of Computer Science
Columbia University
charles@cs.columbia.edu

Henning Schulzrinne
Department of Computer Science
Columbia University
hgs@cs.columbia.edu

Erich Nahum
T.J. Watson Research Center
IBM Research
nahum@watson.ibm.com

Columbia University Computer Science
Technical Report CUCS-031-08

June 2008

Abstract

A SIP server may be overloaded by emergency-induced call volume, "American Idol" style flash crowd effects or denial of service attacks. The SIP server overload problem is interesting especially because the cost of serving and rejecting a SIP session could be in the same neighborhood. For this reason, the built-in SIP overload control mechanism based on generating rejection messages could not prevent the server from entering congestion collapse at heavy load. The SIP overload problem calls for a pushback control solution in which the potentially overloaded receiving server may notify its upstream sending servers to have them send only the amount of load within the receiving server's processing capacity. The pushback framework can be achieved by SIP application layer rate-based feedback or window-based feedback. We propose three new window-based feedback algorithms and evaluate them together with two existing rate-based feedback algorithms. We compare the different algorithms in terms of number of tuning parameters and performance under both steady and dynamic load. Furthermore, we identify two categories of fairness requirements for SIP overload control, namely, user-centric and provider-centric fairness. With the introduction of a new double-feed SIP overload control architecture, we show how the algorithms can meet those fairness criteria.

Contents

1 Introduction

The Session Initiation Protocol [1] (SIP) is a signaling protocol standardized by IETF for creating, modifying, and terminating sessions in the Internet. It has been used for many session-oriented applications, such as voice calls, multimedia distributions, video conferencing, presence service and instant messaging. Major standards bodies including 3GPP, ITU-I, and ETSI have all adopted SIP as the core signaling protocol for Next Generation Networks predominately based on the Internet Multimedia Subsystem (IMS) architecture.

The wide spread popularity of SIP has raised attention to its readiness of countering overload [2]. SIP server can be overloaded for many reasons such as emergency-induced call volume, flash crowds generated by TV programs (e.g., American Idol), special events such as "free tickets to third caller", or even denial of service attacks. Although server overload is by no means a new problem for the Internet, the key observation distinguishes the SIP overload problem from others is that the cost of rejecting a SIP session usually could not be ignored when comparing to the cost of serving a session. Consequently, when a SIP server has to reject a large amount of arriving sessions, its performance collapses. This situation explains the reason why the built-in SIP overload control mechanism based on generating a rejection response messages does not solve the problem. If, as often recommended, the rejected sessions are sent to a load-sharing SIP server, the alternative server will soon also be generating nothing but rejection responses, leading to a cascading failure. Another important aspect of SIP overload is related to its multi-hop server architecture with a name-based application level routing. This aspect creates the so-called "server-to-server" overload problem that is generally not comparable to overload in other servers such as web server.

To avoid the overloaded server ending up at a state spending all its resources rejecting sessions, Hilt *et. al.* [3] outlined a SIP overload control framework based on feedback from the receiving server to its upstream sending servers. The feedback can be in terms of a rate or a load limiting window size. However, the exact algorithms that may be applied in this framework and the potential performance implications are still wide open. In particular, to our best knowledge there has been no published work on specific window-based algorithms for SIP overload control, or comprehensive performance evaluation of rate-based feedback algorithms that also discusses dynamic load conditions and overload control fairness issues.

In this paper, we introduce a new dynamic session estimation scheme which plays an essential role in applying selected control algorithms to the SIP overload environment. We then propose three new window-based algorithms for SIP overload. We also apply two existing load adaptation algorithms for rate-based overload control. Thus we cover all three types of feedback control mechanisms in [3]: the absolute rate feedback, relative rate feedback and window feedback. To evaluate the performance of different algorithms we implemented a SIP simulator on the widely used OPNET platform [4]. Our simulator is one of the independent SIP simulator implementations calibrated in the IETF SIP server overload design team. The simulation results show that although the algorithms differ in their tuning parameters, most of them are able to achieve theoretical maximum performance under steady state load conditions. The results under dynamic load conditions with source arrival and departure are also pretty encouraging. Furthermore, we look at the SIP overload fairness issue and propose the notion of user-centric fairness vs. service-provider-centric fairness. We show how different algorithms may achieve the desired type of fairness. In particular, we found that the user-centric fairness is difficult to achieve in the absolute rate or window-based feedback mechanisms. We solve this problem by introducing a new double-feed SIP overload control architecture.

The rest of this document is organized as follows: Section 2 presents the background on the SIP overload problem, and discusses related work. In Section 3 we propose three window-based SIP overload control algorithms and describe two existing load adaptation algorithm to be applied for rate-based SIP overload control. Then we present the simulation model and basic SIP overload results without feedback control in Section 4. The steady load performance evaluation of the control algorithms are discussed in Section 5, followed by dynamic load performance with fairness consideration in Section 6. Finally Section 7 concludes the paper and talks about future work.

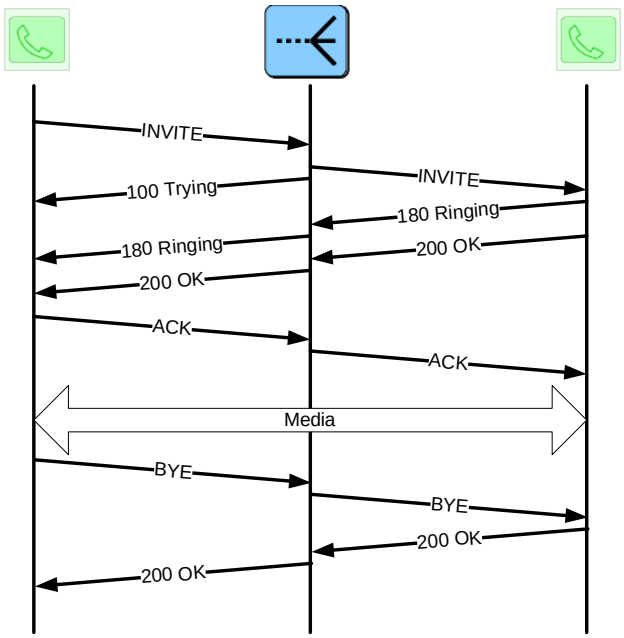

Figure 1: SIP voice call session message flow

2 Background and Related Work

2.1 SIP Overview

SIP is a message based protocol for managing sessions. There are two basic SIP entities, SIP User Agents (UAs), and SIP servers. SIP servers can be further grouped into proxy servers for session routing and registration servers for UA registration. In this paper we focus primarily on proxy servers. In the remainder of this document, when referring to SIP servers, we mean proxy server unless explicitly mentioned otherwise. One of the most popular session type that SIP is used for is voice call session. This is also the type of session we will consider in this paper. In a typical SIP voice call session, the caller and callee has UA functionalities, and they set up the session through the help of SIP servers along the path between them. Figure 1 shows the SIP message flow establishing a SIP call session. The caller starts with sending an INVITE request message towards the SIP proxy server, which replies with a 100 Trying message and forwards the request to the next hop determined by name-based application level routing. In Figure 1 the next hop for the only SIP server is the callee, but in reality it could well be another SIP server along the path. Once the INVITE request finally arrives at the callee, the callee replies with a 180 Ringing message indicating receipt of the call request by the callee UA, and sends a 200 OK message when the callee picks up the phone. The 200 OK message makes its way back to the caller, who will send an ACK message to the callee to conclude the call setup. Afterwards, media may flow between the caller and callee without the intervention of the SIP server. When one party wants to tear down the call, the corresponding UA sends a BYE message to the other party, who will reply with a 200 OK message to confirm the call hang-up. Therefore, a typical SIP call session entails processing of five incoming messages for call setup and two incoming messages for call teardown, a total of seven messages for the whole session.

SIP is an application level protocol on top of the transport layer. It can run over any common transport layer protocols, such as UDP and TCP. A particular aspect of SIP related to the overload problem is its timer mechanism. SIP defines quite a number of retransmission timers to cope with message loss, especially when the unreliable UDP transport is used. As examples we illustrate three of the timers which are commonly seen causing problems under overload. The first is timer A that causes an INVITE retransmission upon each of its expiration. With an initial value of $T_1 = 500\ ms$, timer A increases exponentially until its total timeout

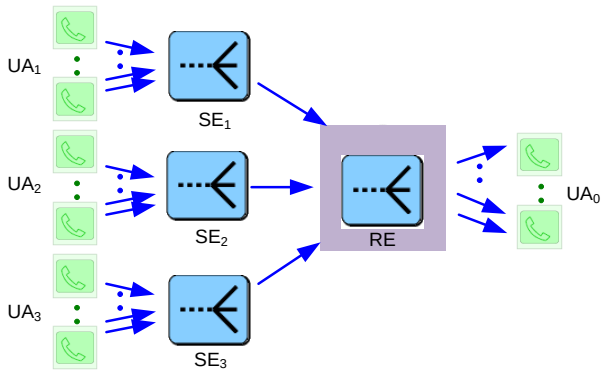

Figure 2: Server to server overload

period exceeds 32 ms. The second timer of interest is the timer that controls the retransmission of 200 OK message as a response to an INVITE request. The timer for 200 OK also starts with T_1, and its value doubles until it reaches $T_2 = 4\ s$. At that time the timer value remains at T_2 until the total timeout period exceeds 32 s. The third timer of interest is timer E which controls the BYE request retransmission. Timer E follows a timeout pattern similar to the 200 OK timer. Note that the receipt of corresponding messages triggered by each of the original messages will quench the retransmission timer. They are the 100 Trying for INVITE, ACK for 200 OK, and 200 OK for BYE. From this description, we know that for example, if an INVITE message for some reason is dropped or stays in the server queue longer than 500 ms without generating the 100 Trying, the upstream SIP entity will retransmit the original INVITE. Similarly, if the round trip time of the system is longer than 500 ms, then the 200 OK timer and the BYE timer will fire, causing retransmission of these messages. Under ideal network conditions without link delay and loss, retransmissions are purely wasted messages that should be avoided.

2.2 Types of SIP Server Overload

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

A typical server-to-server overload topology is illustrated in Figure 2. In this figure the overloaded server (the Receiving Entity or RE) is connected with a relatively small number of upstream servers (the Sending Entities or SEs). One example of server-to-server overload is a special event like "free tickets to the third caller", also referred to as Flash Crowds. Suppose RE is the Service Provider (SP) for a hotline N. SE_1, SE_2 and SE_3 are three SPs that reach the hotline through RE. When the hotline is activated, RE is expected to receive a large call volume to the hotline from SE_1, SE_2 and SE_3 that far exceeds its usual call volume, potentially putting RE into a severe overload.

The second type of overload, known as client-to-server overload, is when a number of clients overload the next hop server directly. An example is avalanche restart, which happens when power is restored after a mass power failure in a large metropolitan area. At the time the power is restored, a very large number of SIP devices boot up and send out SIP registration requests almost simultaneously, which could easily overload the corresponding SIP registration server. This paper only discusses the server-to-server overload problem. The client-to-server overload problem may require different solutions and is out of scope of this paper.

2.3 Existing SIP Overload Control Mechanisms

Without overload control, messages that cannot be processed by the server are simply dropped. Simple drop causes the corresponding SIP timers to fire, and further amplifies the overload situation.

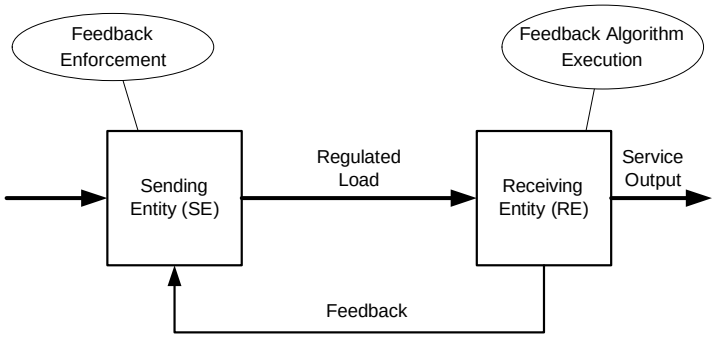

Figure 3: Generalized feedback architecture

SIP has a 503 Service Unavailable response message used to reject a session request and cancel any related outstanding retransmission timers. However, because of the relatively high cost of rejection, this message cannot solve the overload problem.

SIP also defines an optional parameter called "Retry-after" in the 503 Service Unavailable message. The "Retry-after" value specifies the amount of time that the receiving SE of the message should cease sending any requests to the RE. The 503 Service Unavailable with "Retry-after" represents basically an on and off overload control approach, which is known to be unable to fully prevent congestion collapse [2]. Another related technique is to allow the SE to failover the rejected requests to an alternative load-sharing server. However, in many situations the load-sharing server could ultimately be overloaded as well, leading to cascading failure.

2.4 Feedback-based Overload Control

The key to solving the SIP server overload problem is to make sure the upstream SEs only send the amount of traffic that the RE is able to handle at all times. In this ideal situation, there will be no message retransmission due to timeout and no extra processing cost due to rejection. The server CPU power can be fully utilized to deliver its maximum session service capacity.

A feedback loop is a natural approach to achieve the ideal overload control goal. Through the loop, RE notifies SEs the amount of load that is acceptable.

To some extent the existing SIP 503 Service Unavailable with "Retry-after" mechanism is a basic form of the feedback mechanism. Unfortunately its on and off control nature has proven to be problematic. Therefore, the IETF community has started looking at more sophisticated pushback mechanisms including both rate-based and window-based feedback. A generalized model of the feedback-based control model is shown in Figure 3. There are three main components in the model: feedback algorithm execution at RE, feedback communication from RE to SE, and feedback enforcement at the SE.

2.4.1 Feedback Algorithms

Absolute rate, relative rate and window feedback are three main SIP feedback control mechanisms. Each mechanism executes specific control algorithms to generate and adapt the feedback value.

In absolute rate-based feedback, the feedback generation entity RE needs to estimate its acceptable load and allocate it among the SEs. The feedback information is an absolute load value for the particular SE. The key element in absolute rate feedback is an algorithm for dynamic acceptable load estimation.

In relative rate-based feedback, the feedback generation entity RE computes an incoming load throttle percentage based on a target resource metric (e.g., CPU utilization). The feedback information is a dynamic percentage value indicating how much proportion of the load should be accepted or rejected relative to the original incoming load. The key element in relative

rate feedback is the dynamic relative rate adjustment algorithm and the choosing of the target metric.

In window-based feedback, the feedback generation entity RE estimates a dynamic window size for each SE which specifies the number of acceptable sessions from that particular SE. The feedback information is the current window size. The key element in window-based feedback is a dynamic window adjustment algorithm.

The feedback generation could be either time-driven or event-driven. In time-driven control, the control is usually exercised every pre-scheduled control interval, while in event-driven mechanisms, the control is executed upon the occurrence of some events, such as session service completion. We will examine both time-driven and event-driven algorithms in this paper.

2.4.2 Feedback Enforcement

The SEs may choose among many well-known traffic regulation mechanisms to enforce feedback control, such as percentage throttle, leaky bucket and token bucket, automatic call gapping, and window throttle.

Percentage throttle: percentage throttle is usually used with rate-based mechanism. In an absolute rate-based mechanism, assuming the target absolute rate is λ' and the original incoming rate is λ, then SE should block $(1 - \lambda'/\lambda)$ percentage of the requests before sending the load to the RE. In a relative rate-based mechanism, the blocking percentage is usually directly available. The implementation of percentage throttle is also simple. One can generate a random number between 0 and 100 when a request arrives. If the generated number is below the blocking percentage value, the request is rejected, otherwise it is accepted.

Leaky bucket and token bucket: leaky bucket and token bucket control are two classical rate control mechanisms used in the Internet. Both of them provide a long term rate cap, while allow certain traffic burst as specified by the bucket depth. The only difference of these two mechanisms is how the bursts impact the network. While a leaky bucket absorbs the bursts at the application input and smooths them out before passing them to the network, the token bucket passes the burst directly to the network [5]. Both leaky bucket and token bucket mechanisms can be used to enforce rate-based feedback control, although their use with the absolute rate-based mechanism is more straightforward because the absolute rate value directly supplies the most important rate parameter in the two bucket schemes.

Automatic Call Gapping: Automatic call gapping is a technique commonly used in telecommunication networks to regulate call rate. In automatic call gapping, a gap timer is started when receiving a call. All calls subsequently received while the gap timer is still active are blocked. After the gap timer expires, the next call arrival will be accepted and triggers the start of the gap timer again. From its definition, automatic call gap can be seen as a variation of the bucket mechanism with a zero burst size. Comparing to the token bucket and leaky bucket mechanisms, automatic call gapping is more conservative and does not allow burst, the applicability is similar.

Window Throttle: In a window throttle mechanism, the SE keeps track of the current window size as advertised by the RE. If there is a window slot available when a new call arrives at the SE, the call is accepted and forwarded to the RE. Otherwise, the call is blocked by the SE.

A call that is blocked by any of the load regulation mechanisms above may either be queued for future sending opportunities or be rejected immediately, leading to the *keep* vs. *discard* blocking mode of feedback enforcement. Essentially the *keep* mode allows certain level of burst for incoming traffic depending on the regulation queue size. It is worth noting that the *keep* mode is not the necessary condition to allow burstiness. E.g., a token bucket mechanism operating at *discard* blocking mode also permits bursts.

In addition to the burstiness aspect, a more subtle difference between the *keep* mode and the *discard* mode resides in signaling delay. In the *keep* mode, there is certain delay incurred in the load regulation queue which is not present in the *discard* mode. This delay in general does not affect any SIP retransmission timer, but it does contribute to the overall signaling delay and therefore has to be kept within a reasonable value.

Considering the difference between the two modes, the *keep* mode is most suitable when the load regulation contains internal smoothing functions such as in the leaky bucket mechanism. In that case the size of the load regulation queue corresponds to the leaky bucket burst parameter. For other load regulation mechanisms including percentage throttle, token-bucket, automatic call gapping and window throttle, the *discard* mode is usually more convenient. All the above feedback enforcement mechanisms are well-known. Since our focus is on the feedback algorithms, throughout this document we will use percentage throttle for rate-based feedback and window-throttle for window-based feedback mechanisms .

2.4.3 Feedback Communication

The feedback information for SIP signaling overload control can be communicated via an in-band or out-of-band channel. Specifically, an in-band approach may piggyback the feedback information through some new header fields of existing SIP messages; an out-of-band approach may use separate SIP messages such as SIP SUBSCRIBE and NOTIFY messages. In general, the inband approach is more efficient but it does not work well when there are not enough existing messages to carry the feedback information. On the other hand, the outband approach incurs more overhead for generating additional messages, but may carry feedback information anytime necessary.

In this paper, we have chosen to use the in-band feedback communication approach. i.e., any feedback information available is sent in the next immediate message that goes to the particular target SE. This approach fits particularly in the server-to-server overload situation because there is generally no problem finding existing messages to carry feedback information under overload and it incurs minimal overhead.

2.5 Single-hop vs. Multi-hop Pushback

There are two types of pushback control, single-hop vs. multi-hop. In single-hop pushback, a feedback loop is constructed between each pair of the two adjacent SIP servers. Since the SE knows which next hop SIP server is for a received request, it can assess the load condition of the corresponding next-hop SIP server and execute load regulation mechanism for the incoming requests.

On the contrary, multi-hop pushback has the feedback loop extends from the overloaded server to a far upstream SE which may be close to the source of the SIP session. Assuming a upstream SE needs to regulate load for a downstream RE that is multiple hops away. When the upstream SE receives a SIP request, it has to first determine whether the request will eventually go through the specific downstream RE. This is usually not an easy task when there are multiple hops between the SE and the RE and when SIP routing is not strictly static. In addition, when any of the SIP servers between the SE and the RE that forwards the feedback information as part of the multi-hop feedback loop, they have to know the SIP route leading to the specific RE as well. The middle path servers also need to recursively consolidate information from all the next hop SIP servers that lead to the RE.

Because of the restricted SIP routing assumption and the apparent complexity in generating consolidated feedbacks, the multi-hop pushback mechanism is only applicable to some simple network scenarios and is out of the this document.

2.6 Related Work

Signaling overload itself is a well studied topic. Many of the previous work on call signaling overload in general communication networks is believed to be usable by the SIP overload study. For instance, Hosein [6] presented an adaptive rate control algorithm based on estimation of

message queueing delay; Cyr *et. al.* [7] described the Occupancy Algorithm (OCC) for load balancing and overload control mechanism in distributed processing telecommunications systems based on server CPU occupancy; Karsera *et. al.* [8] proposed an improved OCC algorithm called Acceptance-Rate Occupancy (ARO) by taking into consideration the call acceptance ratio, and also a Signaling RED algorithm which is a RED variant for signaling overload control.

Specifically on SIP, Ohta [9] showed through simulation the congestion collapse of SIP server under heavy load and explored the approach of using a priority queueing and Bang-Bang type of overload control. Nahum *et. al.* [10] reported empirical performance results of SIP server showing the congestion collapse behavior.

In addition, Whitehead [11] described a unified overload control framework called GOCAP for next generation networks, which is supposed to cover SIP as well. But there has been no performance results yet and it is not clear at this time how the GOCAP framework may relate to the IETF SIP overload framework.

In the most closely related work to this document, Noel and Johnson [12] presented initial results comparing a SIP network without overload control, with the built-in SIP overload control and with a rate-based overload control scheme. However, their paper does not discuss window-based control, or present performance results under dynamic load, and it does not address the overload fairness problem.

3 Feedback Algorithms for SIP Overload Control

The previous section has introduced the main components of SIP overload feedback control framework. In this section we investigate its key component - the feedback algorithm. We propose three window-based SIP overload control methods, namely *win-disc*, *win-cont*, and *win-auto*. We also apply two existing adaptive load control algorithms for rate-based control. Before discussing algorithm details, we first introduce a dynamic SIP session estimation method which plays an important role in applying selected rate-based or window-based algorithms to SIP overload control.

3.1 Dynamic SIP Session Estimation

Design of SIP overload control algorithm starts with determining the control granularity, i.e., the basic control unit. Although SIP is a message-based protocol, different types of SIP messages carry very different weights from admission control perspective. For instance, in a typical voice call session, admitting a new INVITE message starts a new call and implicitly accepts six additional messages for the rest of the session signaling. Therefore, it is more convenient to use a SIP session as the basic control unit.

A session oriented overload control algorithm frequently requires session related metrics as inputs such as the session service rate. In order to obtain session related metrics a straightforward approach is to do a "*full session check*", i.e., to track the start and end message of all SIP signaling sessions. For example, the server may count how many sessions have been started and then completed within a measurement interval. In the case of a voice call signaling, the session is initiated by an INVITE request and terminated with a BYE request. The INVITE and BYE are usually separated by a random session holding time. However, SIP allows the BYE request to traverse a different server from the one for the original INVITE. In that case, some SIP server may only see the INVITE request while other servers only see the BYE request of a signaling session. There could also be other types of SIP signaling sessions traversing the SIP server.These factors make the applicability of the "*full session check*" approach complicated, if not impossible.

We use an alternative "*start session check*" approach to estimate SIP session service rate . The basic idea behind is that under normal working conditions, the actual session acceptance rate is roughly equal to the session service rate. Therefore, we can estimate the session service rate based only on the session start messages. Specifically, the server counts the number of INVITE messages that it accepts per measurement interval T_m. The value of the session service

rate is estimated to be $\mu = N_{inv}^{accepted}/T_m$. Standard smoothing functions can be applied to the periodically measured μ.

One other critical session parameter often needed in SIP overload control algorithms is the number of sessions remaining in the server system, assuming the server processor is preceded by a queue where jobs are waiting for service. It is very important to recognize that the number of remaining sessions is NOT equal to the number of INVITE messages in the queue, because the queue is shared by all types of messages, including those non-INVITE messages which represent sessions that had previously been accepted into the system. All messages should be counted for the current system backlog. Hence we propose to estimate the current number of sessions in the queue using Eq. 1:

$$N_{sess} = N_{inv} + \frac{N_{noninv}}{L_{sess} - 1} \tag{1}$$

where N_{inv} and N_{noninv} are current number of INVITE and non-INVITE messages in the queue, respectively. The parameter L_{sess} represents the average number of messages per-session. N_{inv} indicates the number of calls arrived at the server but yet to be processed; $N_{noninv}/(L_{sess}-1)$ is roughly the number of calls already in process by the server.

Eq 1 holds for both the "*full session check*" and the simplified "*start session check*" estimation approaches. The difference is how the L_{sess} parameter is obtained. When the "*full session check*" approach is used, the length of each individual session will be counted by checking the start and end of each individual SIP sessions. With our simplified "*start session check*" approach, the session length can be obtained by counting the actual number of messages N_{msg}^{proc}, processed during the same period the session acceptance rate is observed. The session length is then estimated to be $L_{sess} = N_{msg}^{proc}/N_{inv}^{accepted}$.

3.2 Active Source Estimation

In some of the overload control mechanisms, the RE may wish to explicitly allocate its total capacity among multiple SEs. A simple approach is to get the number of current active SEs and divide the capacity equally. We do this by directly tracking the sources of incoming load and maintaining a table entry for each current active SE. Each entry has an expiration timer set to one second.

3.3 The *win-disc* Control Algorithm

A window feedback algorithm executed at the RE dynamically computes a feedback window value for the SE. SE will forward the load to RE only if window slots are currently available. Our first window based algorithm is *win-disc*, the short name for *window-discrete*. The main idea is that at the end of each discrete control interval of period T_c, RE re-evaluate the number of new session requests it can accept for the next control interval, making sure the delays for processing sessions already in the server and upcoming sessions are bounded. Assuming the RE advertised window to SE_i at the k^{th} control interval T_c^k is w_i^k, and the total window size for all SEs at the end of the k^{th} control interval is w^{k+1}, the *win-disc* algorithm is described below:

$w_i^0 := W_0$ where $W_0 > 0$
$w_i^k := w_i^k - 1$ for INVITE received from SE_i
$w^{k+1} := \mu^k T_c + \mu^k D_B - N_{sess}^k$ at the end of T_c^k
$w_i^{k+1} := round(w^{k+1}/N_{SE}^k)$

where μ^k is the current estimated session service rate. D_B is a parameter that reflects the allowed budget message queueing delay. N_{sess}^k is the estimated current number of sessions in the system at the end of T_c^k. $\mu^k T_c$ gives the estimated number of sessions the server is able to process in the T_c^{k+1} interval. $\mu^k D_B$ gives the average number of sessions that can remain in the server queue given the budget delay. This number has to exclude the number of sessions already backlogged in the server queue, which is N_{sess}^k. Therefore, w^{k+1} gives the estimated total number of sessions that the server is able to accept in the next T_c control interval giving delay

budget D_B. Both μ^k and N^k_{sess} are obtained with our dynamic session estimation algorithm in Section 3.1. N^k_{SE} is the current number of active sources discussed in Section 3.2. Note that the initial value W_0 is not important as long as $W_0 > 0$. An example value could be $W_0 = \mu_{eng} T_c$ where μ_{eng} is the server's engineered session service rate.

Note that in this and all following window-based algorithms, although individual window size is maintained for each active upstream SEs, there is one common SIP processing queue shared by all of them. When a message is arrived, the current window availability for the specific SE is always checked. If window slot is available, the message is enqueued. Otherwise, the message may be dropped or rejected. If all SEs implement the window throttle mechanism correctly, there should in general be no drop or rejection due to window slot unavailable.

3.4 The *win-cont* Control Algorithm

Our second window feedback algorithm is *win-cont*, the short name for *window-continuous*. Unlike the time-driven *win-disc* algorithm, *win-cont* is an event driven algorithm that continuously adjusts advertised window size when the server has room to accept new sessions. The main idea of this algorithm is to bound the number of sessions in the server at any time. The maximum number of sessions allowed in the server is obtained by $N^{max}_{sess} = \mu^t D_B$, where D_B is again the allowed message queueing delay budget and μ^t is the current service rate. At any time, the difference between the maximum allowed number of sessions in the server N^{max}_{sess} and the current number of sessions N_{sess} is the available window to be sent as feedback. Depending on the responsiveness requirements and computation ability, there are different design choices. First is how frequently N_{sess} should be checked. It could be after any message processing, or after an INVITE message processing, or other possibilities. The second is the threshold number of session slots to update the feedback. There are two such thresholds, the overall number of available slots W_{ovth}, and the per-SE individual number of available slots W_{indvth}. To make the algorithm simple, we choose per-message processing N_{sess} update and we fix both W_{ovth} and W_{indvth} to 1. unless the values need to be changed for comparison purpose. A general description of the *win-cont* algorithm is summarized as below:

$w^0_i := W_0$ where $W_0 > 0$
$w^t_i := w^t_i - 1$ for INVITE received from SE_i
$w^t_{left} := N^{max}_{sess} - N_{sess}$ upon msg processing
$if(w^t_{left} \geq 1)$
    $w^t_{share} = w^t_{left}/N^t_{SE}$
        $w^t_{i'} := w^t_{i'} + w^t_{share}$
    $if(w^t_{i'} \geq 1)$
        $w^t_i := (int)w^t_{i'}$
        $w^t_{i'} := (frac)w^t_{i'}$

Note that since w^t_i may contain a decimal part, to improve the feedback window accuracy when w^t_i is small, we feedback the integer part of the current w^t_i and add its decimal part to the next feedback by using a temporary parameter $w^t_{i'}$.

In the algorithm description, μ^t, N_{sess} and N_{SE} are obtained as discussed in Section 3.1 and Section 3.2. The initial value W_0 is not important and a reference value is $W_0 = \mu_{eng} T_c$ where μ_{eng} is the server's engineered session service rate.

3.5 The *win-auto* Control Algorithm

Our third window feedback algorithm, *win-auto* stands for *window-autonomous*. Like *win-cont* , *win-auto* is also an event driven algorithm. But as the term indicates, the *win-auto* algorithm is able to make window adjustment autonomously. The key design principal in the *win-auto* algorithm is to automatically keep the pace of window increase below the pace of window decrease, which makes sure the session arrival rate does not exceed the session service rate. The algorithm details are as follows:

$w_i^0 := W_0$ where $W_0 > 0$
$w_i^t := w_i^t - 1$ for INVITE received from SE_i
$w_i^t := w_i^t + 1$ after processing a *new* INVITE

The beauty of this algorithm is its extreme simplicity. The algorithm takes advantage of the fact that retransmission starts to occur as the network gets congested. Then the server automatically freezes its advertised window to allow processing of backlogged sessions until situation improves. The only check the server does is whether an INVITE message is a retransmitted one or a new one, which is just a piece of normal SIP parsing done by any existing SIP server. There could be many variations along the same line of thinking as this algorithm, but the one as described here appears to be one of the most natural options.

3.6 The *rate-abs* Control Algorithm

We implemented an absolute rate feedback control by applying the adaptive load algorithm of Hosein [6], which is also used by Noel [12]. The main idea is to ensure the message queueing delay does not exceed the allowed budget value. The algorithm details are as follows.

During every control interval T_c, the RE notifies the SE of the new target load, which is expressed by Eq.2.

$$\lambda^{k+1} = \mu^k(1 - \frac{(d_q^k - D_B)}{C}) \tag{2}$$

where μ^k is the current estimated service rate and d_q^k is the estimated server queueing delay at the end of the last measurement interval. It is obtained by $d_q^k = N_{sess}/\mu^k$, where N_{sess} is the number of sessions in the server. We use our dynamic session estimation in Section 3.1 to obtain N_{sess}, and we refer to this absolute rate control implementation as *rate-abs* in the rest of this document.

3.7 The *rate-occ* Control Algorithm

Our candidates of existing algorithms for relative rate based feedback control are Occupancy Algorithm (OCC) [7], Acceptance-Rate Occupancy (ARO), and Signaling RED (SRED) [8]. We decided to implement the basic OCC algorithm because this mechanism already illustrates inherent properties with any occupancy based approach. On the other hand, tuning of RED based algorithm is known to be relatively complicated.

The OCC algorithm is based on a target processor occupancy, defined as the percentage of time the processor is busy processing messages within a measurement interval. So the target processor occupancy is the main parameter to be specified. The processor occupancy is measured every measurement interval T_m. Every control interval T_c the measured processor occupancy is compared with the target occupancy. If the measured value is larger than the target value, the incoming load should be increased. Otherwise, the incoming load should be decreased. The adjustment is reflected in a parameter f which indicates the acceptance ratio of the current incoming load. f is therefore the relative rate feedback information and is expressed by the Eq. 3:

$$f^{k+1} = \begin{cases} f_{min}, & \text{if } \phi^k f^k < f_{min} \\ 1, & \text{if } \phi^k f^k > 1 \\ \phi^k f^k, & \text{otherwise} \end{cases} \tag{3}$$

where f_k is the current acceptance ratio and f_{k+1} is the estimated value for the next control interval. $\phi^k = min(\rho_B/\rho_t^k, \phi_{max})$. f_{min} exists to give none-zero minimal acceptance ratio, thus prevents the server from completely shutting off the SE. ϕ_{max} defines the maximum multiplicative increase factor of f in two consecutive control intervals. In this report we choose the two OCC parameters ϕ_{max} and f_{min} to be 5 and 0.02, respectively in all our tests.

We will refer to this algorithm as *rate-occ* in the rest of this document.

4 Simulation Model and Basic Performance

4.1 Simulation Platform

We have built a SIP simulator on the popular OPNET modeler simulation platform. Our SIP simulator captures both the INVITE and non-INVITE state machines as defined in RFC3261. It is also one of the independent implementations in the IETF SIP server overload design team, and has been verified in design team under common simulation scenarios.

Our general SIP server model consists of a FIFO queue followed by a SIP processor. Depending on the control mechanisms, specific overload related pre-queue or post-queue processing may be inserted, such as window increase and decrease mechanisms. The feedback information is included in a new *overload* header of each SIP messages, and are processed along with normal SIP message parsing. Processing of each SIP messages creates or updates transaction states as defined by RFC3261. The transport layer is UDP, and therefore all the various SIP timers are in effect.

Our UA model mimics an infinite number of users. Each UA may generate calls at any rate according to a specified distribution and may receive calls at any rate. The processing capacity of UA is assumed to be infinity since we are interested in the server performance.

4.2 Simulation Topology and Configuration

We use the topology in Figure 2 for current evaluations. There are three UAs on the left, each represents infinite number of callers. Each UA is connected to an SE. The three SEs all connect to the RE which is the potentially overloaded server. The queue size is 500 messages. The core RE connects to UA_0 which represents infinite number of callees. Calls are generated with exponential inter-arrival from the callers at the left to the callees on the right. Each call signaling contains seven messages as illustrated in Figure 1. The call holding time is assume to be exponentially distributed with average of 30 seconds. The normal message processing rate and the processing rate for rejecting a call at the RE are 500 messages per second (mps) and 3000 mps, respectively.

Note that the server processer configuration, together with the call signaling pattern, results in a nominal system service capacity of 72 cps. All our load and goodput related values presented below will be normalized to this system capacity. Our main result metric is goodput, which counts the number of calls with successful delivery of all five call setup messages from INVITE to ACK below 10 s.

For the purpose of this simulation, we also made the following assumptions. First: we do not consider any link transmission delay or loss. However, this does not mean feedback is instantaneous, because we assumed the piggyback feedback mechanism. The feedback will only be sent upon the next available message to the particular next hop. Second, all the edge proxies are assumed to have infinite processing capacity. By removing the processing limit of the edge server, we avoid the conservative load pattern when the edge proxy server can itself be overloaded.

These simple yet classical network configuration and assumptions allow us to focus primarily on the algorithms themselves without being distracted by factors of less importance, which may be further explored in future work.

4.3 SIP Overload Without Feedback Control

For comparison, we first look at SIP overload performance without any feedback control.

Figure 4 shows the simulation results in two basic scenarios. In the "Simple Drop" scenario, any message arrived after the queue is full are simply dropped. In the "Threshold Rejection" scenario, the server compares its queue length with a high and a low threshold value. If the queue length reaches the high threshold, new incoming INVITE requests will be rejected but other message are still processed. The processing of new INVITE requests will not be restored until the queue length falls below the low threshold. As we can see, the two result goodput curves almost overlap. Both cases display similar precipitous drop when the offered approximates the

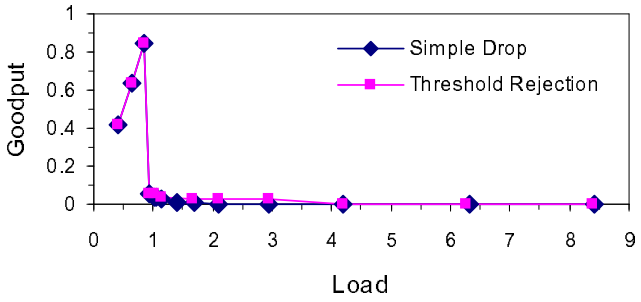

Figure 4: SIP overload with no feedback control

server capacity, a clear sign of congestion collapse. However, the reasons for the steep collapse of the goodput are quite different in the two scenarios. We use the offered load of 450 cps as an example to illustrate more details in both cases which are shown in Figure 5 and Figure 6. For easy display, the curves are moving average smoothed over 60 seconds.

From Figure 5 and Figure 6 we can see that, a bit contrary to intuition, in the "Simple Drop" mechanism, one third of all calls are still delivered to and responded by the callee, while with "Threshold Rejection" no calls could reach the callee. Further exploring Figure 5, we found the following problems associated with that scenario. Since all messages are competing resources with the same priority, the volume of a particular type of the message determines its success probability. The 180 RINGING messages does not retransmit so it has the lowest volume and almost got completed dropped. The 200 OK messages retransmission mechanism gives itself ten times of the volume of 180 RINGING. The INVITE messages has three fewer retransmission opportunities per message compared with the retransmission opportunities of 200 OK. But its original volume is much larger than the original volume of 200 OK since the number of 200 OK messages is only the fraction of the INVITE that arrive at the callee. Overall, the INVITE volume is still clearly larger than the 200 OK volume. (Note that if there are more than one congested server in the path, the INVITE message will have an additional advantage over 200 OK in getting through because of its hop-by-hop reliability delivery vs. the 200 OK's end-to-end reliability delivery.) Figure 5 shows both INVITE and 200 OK only have a very small portion of their messages got processed.

For the "Threshold Rejection" case, the reason no calls are delivered to the callee is that the queue size is constantly at the edge of full occupancy. Therefore, almost all INVITE messages are either rejected or dropped. The server virtually spends all its time rejecting calls.

There are various quick improvements we can make to the above two scenarios given our observation. For example, for "Simple Drop", we could assign higher priority for 200 OK, or release the assumption of mandatory receipt of 180 RINGING message. However, our simulation with those fixes show virtually no improvements in performance because the server is still always flooded with much higher load then it can handle. For "Threshold Rejection", one fix would be to place an artificial cap on the fraction of CPU power that could be used for rejecting calls. But this would not work out because excessive calls beyond the server's rejection capacity would still be dropped and those dropped calls would introduce a retransmission flood that paralyzes the server.

5 Steady Load Performance

5.1 Parameter Summary for Different Feedback Algorithms

We summarize in table 1 the parameters for all rate-based and window-based overload control algorithms we discussed in Section 3. In essence most of the algorithms have a "binding" parameter, three of them use the budget queueing delay D_B, and one uses the budget CPU

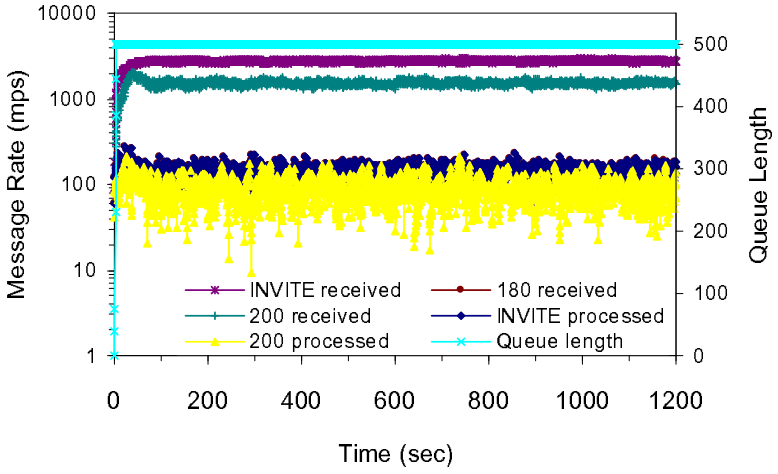

Figure 5: "Simple Drop" mechanism

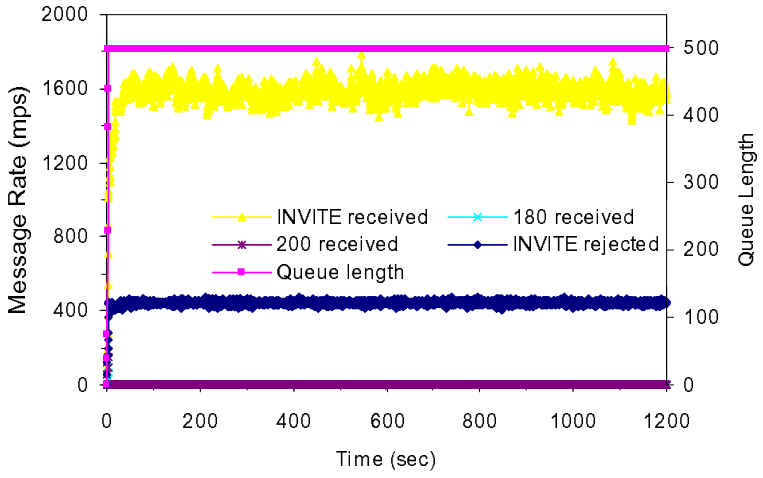

Figure 6: "Threshold Rejection" mechanism

Table 1: Parameter sets for overload algorithms

Algorithm	Binding	Control Interval	Measure Interval	Additional
rate-abs	D_B	T_c	T_m	
rate-occ	ρ_B	T_c	T_m	f_{min} and ϕ
win-disc	D_B	T_c	T_m	
win-cont	D_B*	N/A	T_m	
win-auto	N/A†	N/A	N/A	

D_B: budget queueing delay
ρ_B: CPU occupancy
T_c: discrete time feedback control interval
T_m: discrete time measurement interval for selected server metrics;
f_{min}: minimal acceptance fraction
ϕ: multiplicative factor
* D_B recommended, fixed binding window size also possible
† Optionally D_B may be applied for corner cases

occupancy ρ_B. All three discrete time control algorithms have a control interval parameter T_c.

5.2 The *win-disc* Control Algorithm

We looked at the sensitivity of D_B and T_c for each applicable algorithms. Figure 7 and Figure 8 show the results for *win-disc*. All the load and goodput values have been normalized upon the theoretical maximum capacity of the server.

We started with a T_c value of 200 ms and found that the server achieves the unit goodput when D_B is set to 200 ms. Other $0 < D_B <$200 ms values also showed similar results. This is not surprising given that both the SIP caller INVITE and callee 200 OK timer starts at T_1 = 500 ms. If the queueing delay is smaller than $(1/2)T1$ or 250 ms, then there should be no timeout either on the caller or callee side. A larger value of D_B triggers retransmission timeouts which reduces the server goodput. For example, Figure 7 shows that at D_B = 500 ms, the goodput has already degraded by 25%.

Letting D =200 ms, we then looked at the influence of T_c. As expected, the smaller the value of T_c the more accurate the control would be. In our scenario, we found that a T_c value smaller than 200 ms is sufficient to give the theoretical maximum goodput. A larger T_c quickly deteriorates the results as seen from Figure 8.

5.3 The *win-cont* Control Algorithm

The *win-cont* algorithm has one main parameter which is the budget queueing delay D_B. The goodput results under increasing load at different values of D_B is shown in Figure**??**. According to the algorithm, D_B should be larger than zero. Because the event-driven *win-cont* algorithm is more accurate and finer grained in window control than *win-disc*. The range of D_B values that achieves theoretical maximum goodput expands to 400 ms, which is wider than the *win-disc* case. It should be noted that when D_B is set to 400 ms, the server is still at the theoretical maximum goodput. That means the budget queueing delay is not fully used. In this particular case, the actual server queuing delay is only around 100 ms, thus prevents the SIP retransmission timer from firing.

Although *win-cont* is not time-driven and does not have the notion of an explicitly defined fixed control interval, we can simulate different control frequency by changing the overall available window slot update threshold W_{ovth}, as shown in Figure 10. Since one window slot corresponds to around 14 ms, the W_{ovth} value of 1, 2, 7, 14 correspond to roughly 14 ms, 28 ms,

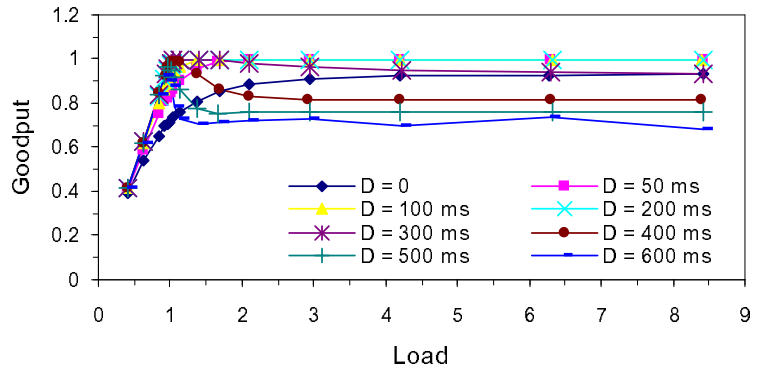

Figure 7: *win-disc* goodput under different queuing delay budget

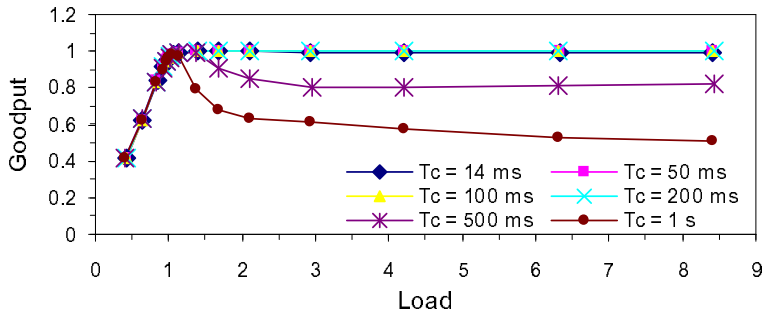

Figure 8: *win-disc* goodput under different control interval T_c

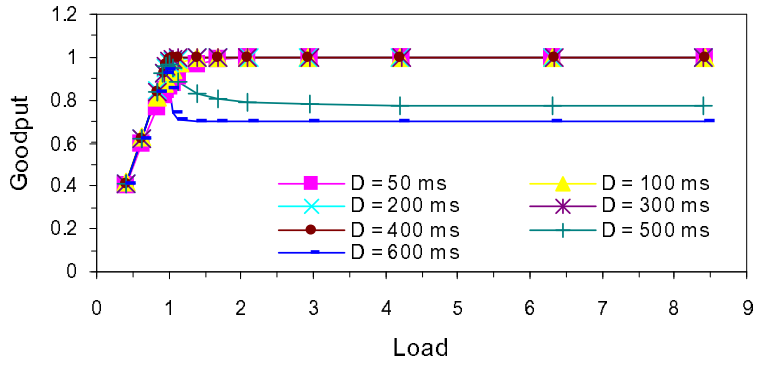

Figure 9: *win-cont* goodput under different budget queuing delay

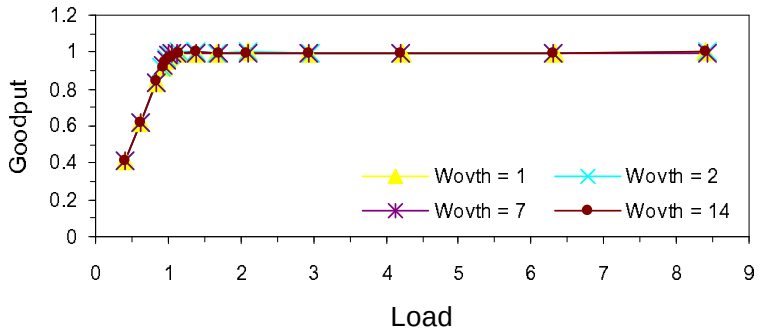

Figure 10: *win-cont* goodput under different W_ovth

98 ms, 196 ms. It can be seen that the performance remains at the theoretical maximum in all cases, both at light and heavy overload.

5.4 The *win-auto* Control Algorithm

The *win-auto* algorithm has virtually no parameters during normal operation. In evaluating the steady state performance, we do need to supply an initial window value. We found that the choice of appropriate initial window depends on the load pattern. If we consider the extreme case where the load pattern is a step function, the choices of the initial window parameter are described below.

If the steady load target of the step function is below the server capacity, a window size that is very small (e.g. one) will render a goodput somewhat below the maximum. There are two ways to solve this problem: one is to disable the window-control when the load is below the server capacity. This introduces an additional feedback activation threshold parameter. The other approach is simply to supply a relatively larger initial window size. We found that a value in the order of ten works well.

If the steady load target exceeds the server capacity, am extremely small window size (e.g., 1) is almost always sufficient to result in an ideal maximum goodput. A larger initial window size in the order of ten in this situation may sometimes lead to suboptimal performance of not reaching the maximum theoretical goodput. Further investigation reveals that, this suboptimal performance is caused by the difference in stabilized queueing delay. If the initial window size is one, when the system reaches steady state, the queueing delay is very small compared to the 500 ms SIP T1 timer. If a larger initial window is applied, however, the system may stabilizng at a point where the queueing delay can exceed 250 ms. The round-trip delay then exceeds 500 ms, which triggers the 200 OK and the BYE retransmission timer, both fire at 500 ms. The two timer expiration introduce three additional messages to the system, a retransmitted 200 OK, the ACK to the retransmitted 200 OK, and a retransmitted BYE. This situation increases the normal session length from seven to ten and reduces the maximum server goodput by 28%. A cure to this situation is to introduce an extra queueing delay binding parameter to the window adjustment algorithm. Specifically, before the server increases the window size, it checks the current queueing delay, if the queueing delay value already exceeds the desired threshold, the window is not increased. However, the optimal value of the queueing delay threshold parameter is not very straightforward and the introducing of the additional parameter leads to much more complexity. Figure 11 shows an example. At a load of 8.3, the original goodput of the *win-auto* algorithm is about 0.7, which is sub-optimal. When the additional delay check is applied and $D = 200\ ms$, the system reaches theoretical goodput of 1. In other cases when $D = 100\ ms$ or $D = 300\ ms$ the goodput does not offer as much improvement.

With all the above considerations for choosing the initial window value, it is important to keep in mind that in realistic scenarios the decision is much easier and the sub-optimal behavior

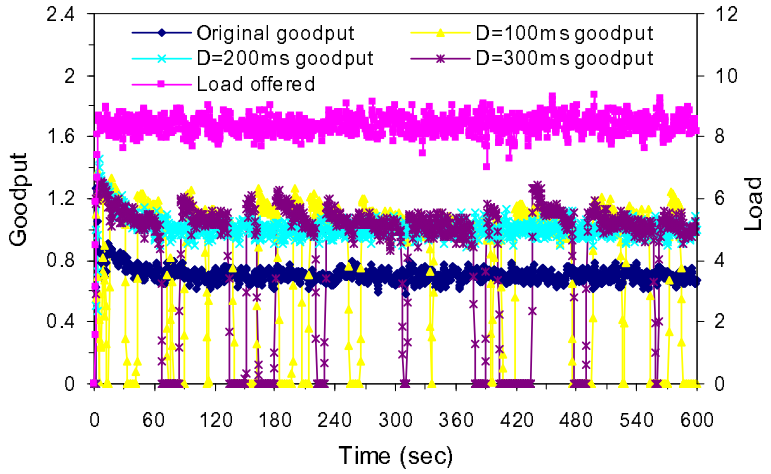

Figure 11: The *win-auto* sub-optimal goodput case

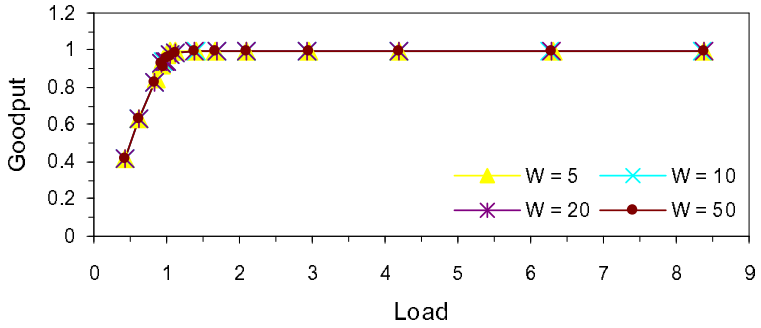

Figure 12: The *win-auto* goodput under different initial window size

can mostly be avoided without the added delay binding. Normally loads are increased gradually when a server starts. Choosing an initial window value in the order of ten is usually appropriate. Once the startup phase is finished, the server window size stabilizes at a pretty small value and operates as a zero-parameter autonomous system close to its maximum capacity. Figure 12 shows the results of *win-auto* under steady loads, all the loads have an increase rate of two sessions per second until it reaches the specified stable rate. It is seen to achieve maximum theoretical goodput at all the four different initial window size of 5, 10, 20, 50.

5.5 The *rate-abs* Control Algorithm

The two main parameters of the *rate-abs* algorithm are control interval T_c and budget queuing delay D_B. The goodput of *rate-abs* under different budget queuing delay when $T_c = 200\ ms$ is shown in Figure 13. The effects of T_c given with budget queueing delay $D = 200\ ms$ is plotted in Figure 14 The sensitivity of *rate-abs* to the two parameters T_c and D is less than *win-disc*, and overall performance is in between *win-auto* and *win-disc*. This could be partially attributed to the fact that rate-abs employees a percentage-throttle at the SE, which result in less bursty of the traffic sending to the *RE* compared to *win-disc*. On the other hand, the event driven *win-cont* offers a more accurate than the time-driven algorithms.

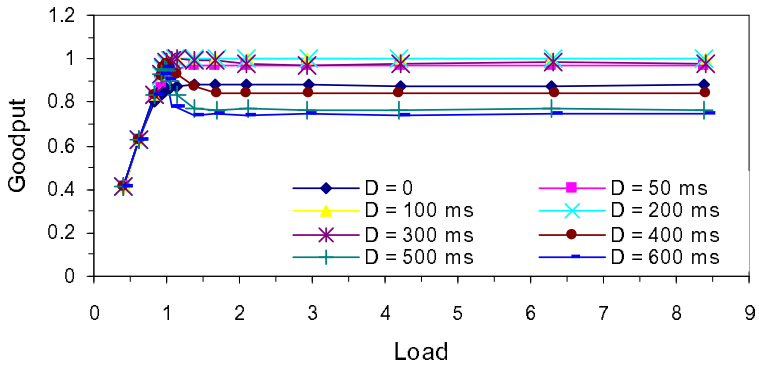

Figure 13: *rate-abs* goodput under different queuing delay budget

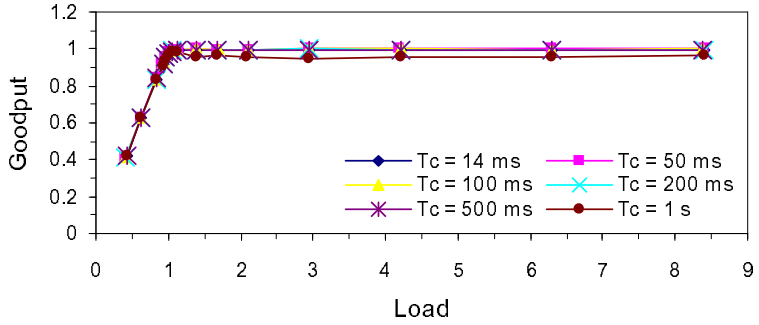

Figure 14: *rate-abs* goodput under different control interval

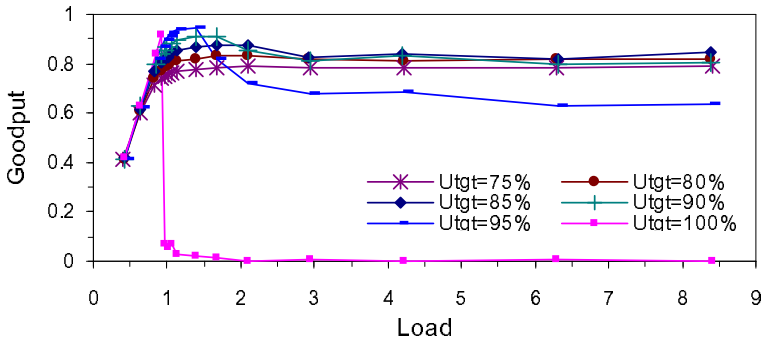

Figure 15: *rate-occ* goodput under different U_{tgt}

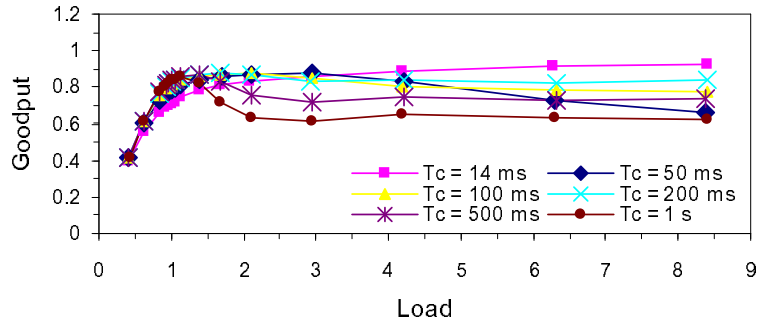

Figure 16: *rate-occ* goodput under different Tc

5.6 The *rate-occ* Control Algorithm

The *rate-occ* algorithm has a number of parameters ϕ_{max}, f_{min}, U and T_c. We set $\phi_{max} = 5$, $f_{min} = 0.02$, $T_c = 200\ ms$ and plot the goodput under different values of ρ_B in Figure 15. As it shows, under the specific setting, $\rho_B = 85\%$ gives the best goodput performance which is about 0.85, matching the CPU occupancy level. $\rho_B = 100\%$ results in virtually no control. For other values, the higher the ρ_B the better the performance for the light overhead immediately after passing the server saturation point, but the stable goodput drops to 0.6 - 0.7 when $\rho_B = 95\%$.

Figure 15 shows *rate-occ* under different control intervals given the same ϕ_{max}, f_{min} and with ρ_B set to 85%. When Tc increases to over 200 ms, the stable goodput during overload decreases as expected. What is particularly interesting in this figure is that unlike *rate-abs* and *win-disc* where lower T_c always result in better performance, in *rate-occ* the performance under a short control interval is more complicated. For example, the results of $T_c = 50\ ms$ show some degradation in the stable goodput under heavy load compared to $T_c = 200\ ms$. But at $T_c = 14\ ms$, the results show a surprising up trend at the heavy overload region over load of 4, although The goodput of the light overload region below load of 1.4 is poorer than all the rest scenarios. These are caused by the multiplicative increase/decrease nature of the algorithm, and the choice of the multiplicative increase/decrease factor would affect the exact behavior. In fact, when the $T_c = 14\ ms$, the algorithm is no longer working in the close region of 85% CPU occupancy as originally desired, instead the CPU occupancy reaches 93% at the load of 8.4.

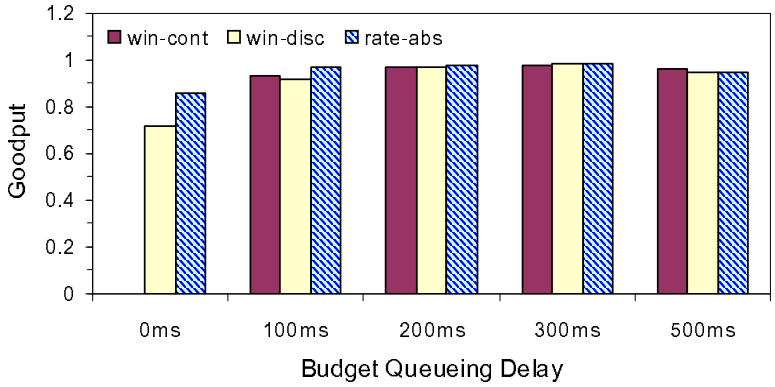

Figure 17: Goodput vs. D_B at load value 1

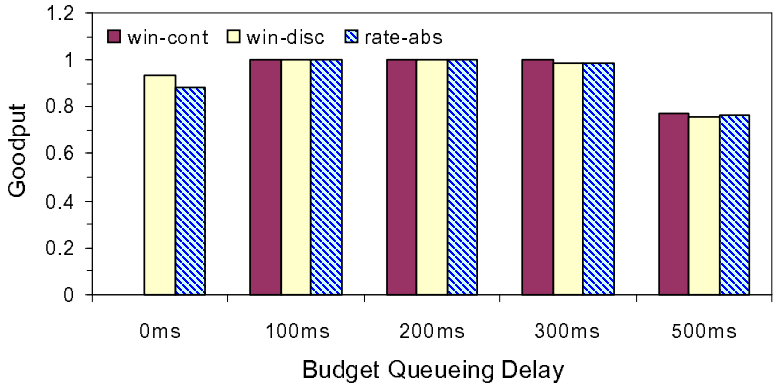

Figure 18: Goodput vs. D_B at load value 8.4

5.7 Comparing Performance of the Different Algorithms

There are two interval parameters T_m and T_c used by a subset of the algorithms. T_m is used by four of the five algorithms. T_m and T_c need to be separate only when T_c is relatively large compared to T_m. The choice of the T_m value depends on how volatile the target server metric is over time. For example, if the target metric is the server service rate, which is relatively stable, a value of 100 ms is usually more than sufficient. If on the other hand, the target metric is the current queue length, then smaller or larger T_m makes clear differences. In our study, when the specific algorithm requires to measure the server service rate and CPU occupancy, we apply T_m; when the algorithm requires information on the current number of packets in the queue, we always obtain the instant value. Our results show that $T_m = min(100\ ms, T_c)$ is a reasonable assumption, by which we reduce the two interval parameters into one.

In Figure 17 and Figure 18, we plotted the goodput results from three algorithms *win-cont*, *win-disc* and *rate-abs* under different D_B values under two different load conditions: at the edge of overload where load is 1, and a heavy overload where load is 8.4.

Since the *win-cont* algorithm does not allow D_B to be zero, the value for *win-cont* at 0 *ms* is not reported. For other D_B values, all three algorithms perform very similar. The only difference for *win-disc* and *rate-abs* is at $D_B = 0$, where *rate-abs* is better than *win-disc* at light overload. The D_B parameter essentially provides some bursty capability at the server. So this could be related to the fact that the window throttle mechanism used by *win-disc* results in a more busty arrival at the server than the percentage throttle mechanism in *rate-abs*. In any case, it is clear that a positive D_B value centered at around 200 ms provides a good outcome.

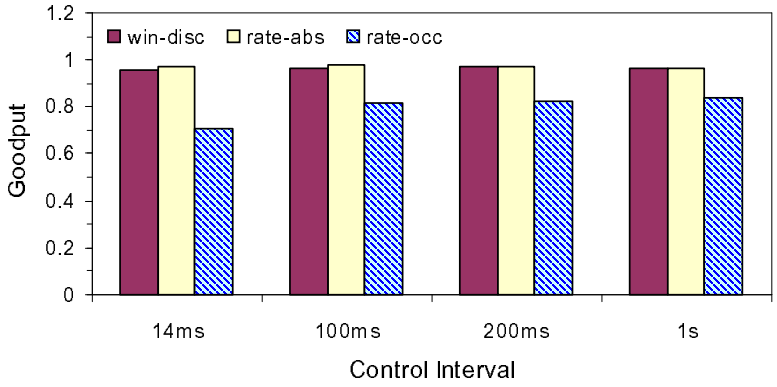

Figure 19: Goodput vs. T_c at load 1

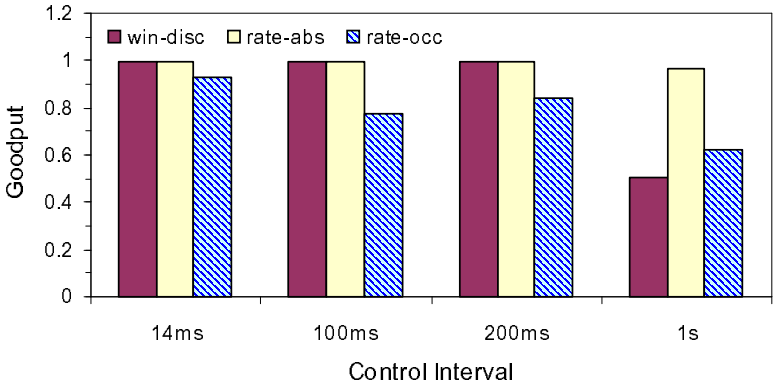

Figure 20: Goodput vs. T_c at load 8.4

Table 2: Parameters used for comparison

	$D_B(ms)$	$T_c(ms)$	$T_m(ms)$
rate-abs	200	200	100
rate-occ1‡	N/A	200	100
rate-occ2‡	N/A	14	14
win-disc	200	200	100
win-cont	200	N/A	100
win-auto	N/A	N/A	N/A

‡ in addition: $\rho_{tg} = 0.85, \phi = 5, f_{min} = 0.02$

Figure 19 and Figure 20 compare the T_c parameter for *win-disc*, *rate-abs* and *rate-occ* with $B_D = 200\ ms$. For the *rate-occ* binding parameter ρ_B, we used 85% for the tests in Figure 19 and Figure 20. We will explain why this value is chosen shortly. It can be seen that the performance of *win-disc* and *rate-abs* are very close to maximum theoretical value in all cases except for when $T_c = 1\ s$ in the heavy load case. This shows *win-disc* is more sensitive to control interval than *rate-abs*, which could also be caused by the more busty nature of the traffic resulted from window throttle. It is clear that for both *win-disc* and *rate-abs* a shorter T_c improves the results, and a value below 200 ms is sufficient. Overall, *rate-occ* performs not as good as the other two. But what is interesting about *rate-occ* is that from 14 ms to 100 ms control interval, the goodput increases in light overload and decreases in heavy overload. This could be a result of rate adjustment parameters which may have cut the rate too much at the light overload.

Having looked at various parameters for all different algorithms, we now summarize the best goodput achieved by each algorithm in Figure 21. The specific parameters used for each algorithm is listed in Table 2.

It is clear from Figure 21 that all algorithms except for *rate-occ* are able to reach the theoretical maximum goodput. The corresponding CPU occupancy also confirms the goodput behavior. What is important to understand is that the reason *rate-occ* does not operate at the maximum theoretical goodput like the others is not simply because of the artificial limit of setting the occupancy to 85%. This point can be confirmed by the earlier Figure 15. The inherent issue with an occupancy based heuristic is the fact that occupancy is not as direct a metric as queue length or queueing delay in solving the overload problem. Figure 21 shows one factor that really helps improve the *rate-occ* performance at heavy load seem to be using extremely small T_c. But updating the current CPU occupancy every 14 ms is not straightforward in all systems. Furthermore, when this short T_c is used, the actual server occupancy rises to 93%, which goes contrary to the original intention of setting the 85% budget server occupancy. Yet another issue with setting the extremely short T_c is its much poorer performance than other algorithms under light overload, which should be linked to the tuning of OCC's heuristic increase and decrease parameters.

The merits of all the algorithms achieving maximum theoretical goodput is that they ensure no retransmission ever happens, and the server is thus always busy processing messages, with each single message being part of a successful session.

Another metric of interest for comparison is the session setup delay, which we define as from the time the INVTE is sent until the ACK to 200 OK message is received. We found that the *rate-occ* algorithm has the lowest delay but this is not significant considering it operates at the sub-optimal region in terms of goodput. *win-cont* comes next with a delay of around 3 ms. The *rate-abs* offers a delay close to that of *win-cont* at about 3.5 ms. The remaining two *win-disc* and *win-auto* have a delay of 5 ms and 6 ms respectively. In fact all these values are sufficiently small and are not likely make any difference.

From the steady state load analysis so far, we conclude that the occupancy based approach is less favorable than others because of its relatively more number of tuning parameters and not being able to adapt to the most efficient processing condition for the maximum goodput. *win-disc* and *abs-rate* are by definition quite similar and they also have the same number of parameters. Their performance are also very close, although *rate-rate* has shown a slight edge,

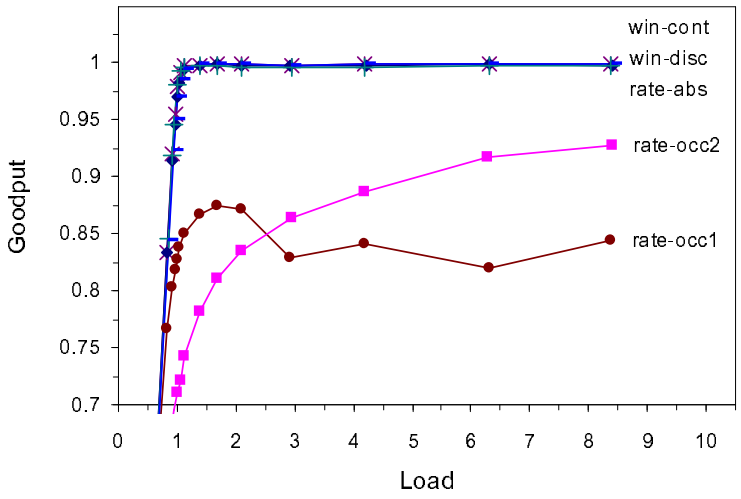

Figure 21: Goodput performance for different algorithms

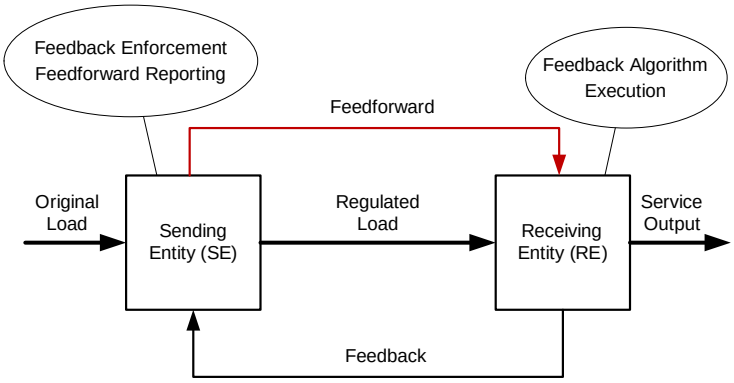

Figure 22: The double feed architecture

possibly because of the smoother arrival pattern resulted from percentage throttle. *win-cont* has less tuning parameter than *win-disc* and *abs-rate*, and offers equal or slightly better performance Finally, *win-auto* is an extremely simple algorithm yet achieves perfect results in most situations.

6 Dynamic Load Performance and Fairness for SIP Overload Control

Although steady load performance is a good starting point for evaluating the overload control algorithms, most of the regular overload scenarios are not persistent steady overload. Otherwise, the issue would become a poor capacity planning problem. The realistic server to server overload situations are more likely short periods of bulk loads, possibly accompanied by new sender arrivals or departures. Therefore, in this section we extend our evaluation to the dynamic behavior of overload control algorithms under load variations. Furthermore, we investigate the fairness property of each of the algorithms.

6.1 Fairness for SIP Overload Control

6.1.1 Defining Fairness

Under overload, the server may allocate its available capacity among all the upstream senders using criteria considered fair. Theoretically, fairness can be coupled with many other factors and could have unlimited number of definitions. However, we see two basic types of fairness criteria which may be applicable in most scenarios: service provider-centric and end-user-centric.

If we consider the upstream servers representing service providers, a service-provider centric fairness means giving all the upstream servers the same aggregate success rate.

The user-centric fairness criteria aim to give each individual user who are using the overloaded server the same chance of call success, regardless of where the call originated from. Indeed, this end-user-centric fairness may be preferred in regular overload situation. For example, in the TV hotline "free tickets to the third caller" case, user-centric fairness ensures that all users have equal winning probability to call in. Otherwise, a user with a service provider who happens to have a large call volume would be in a clear disadvantage.

6.1.2 Achieving Fairness

Technically, achieving the basic service provider-centric fairness is easy if the number of active sources are known, because the overloaded server simply needs to split its processing capacity equally in the feedback generated for all the active senders.

Achieving user-centric fairness means the overloaded server should split is capacity proportionally among the senders based on the senders original incoming load. For the various feedback mechanisms we have discussed, technically the receiver in both the absolute rate-based and window-based feedback mechanisms does not have the necessary information to do proportional capacity allocation when the feedback loop is activated. The receiver in the relative rate-based mechanism does have the ability to deduce the proportion of the original load among the senders.

To achieve user-centric fairness in absolute rate and window-based mechanisms, we introduce a new feedforward loop in the existing feedback architecture. The resulting double-feed architecture is shown in Figure 22. The feedforward information contains the sender measured value of the current incoming load. Like the feedback, all the feedforward information is naturally piggybacked in existing SIP messages since SIP messages by themselves travel in both directions. This way the feedforward introduces minimal overhead as in the feedback case. The feedforward information from all the active senders gives the receiver global knowledge about the original sending load. It is worth noting that, this global knowledge equips the receiver with great flexibility that also allows it to execute any kind of more advanced user-centric or service provider-centric fairness criteria. Special fairness criteria may be required, for example, when the server is experiencing denial of service attack instead of regular overload.

6.2 Dynamic Load Performance

Figure 23 depicts the arrival pattern for our dynamic load test. We used the step function load pattern because if the algorithm works in this extreme case, it should work in less harsh situations. The three UAs each starts and ends at different time, creating an environment of dynamic source arrival and departure. Each source also has a different peak load value, thus allowing us to observe proportional fairness mechanisms when necessary.

Figure 24, Figure 25 and Figure 26 show the performance of *rate-abs*, *win-disc* and *win-cont* under dynamic load. In all cases, the goodput follows well with the change of total offered load. As long as the total offered load exceed the server capacity, the total goodput is around 1.

The results also show that in the *rate-abs* and *win-dsic* case, when the total offered load exceeds the server capacity, the goodput shared by each active source is roughly equal, achieving provider-centric fairness. In the case of *win-cont*, the share of allocation is not clear. This means that the basic algorithm in Section 5.3, although intended to achieve equal capacity for each source, does not actually guarantee that. However, we will present below an improved *win-cont* algorithm which can solve this problem.

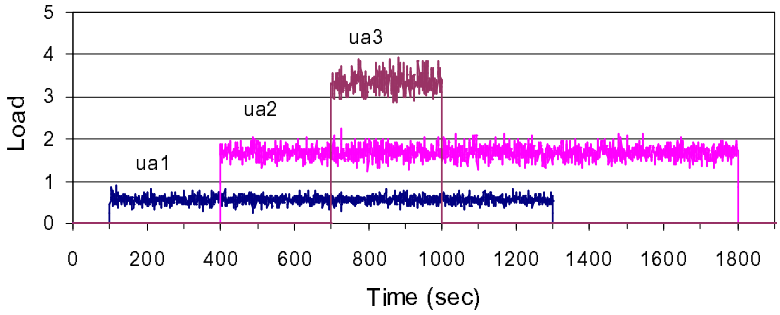

Figure 23: Dynamic load arrival

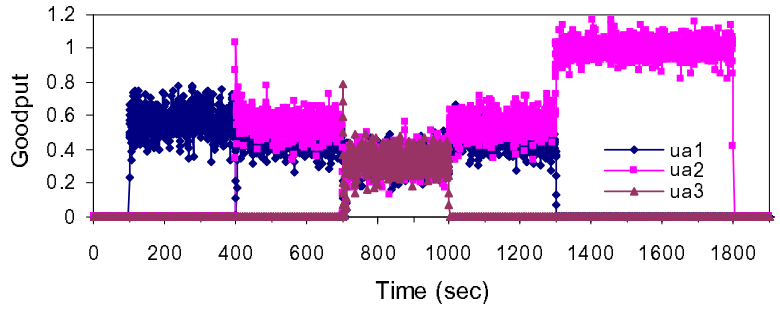

Figure 24: *rate-abs* goodput under dynamic load

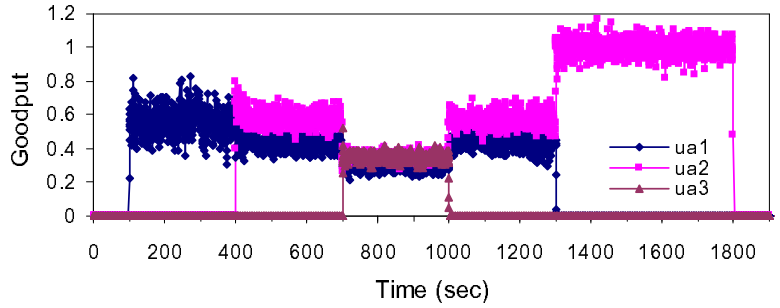

Figure 25: *win-disc* goodput under dynamic load

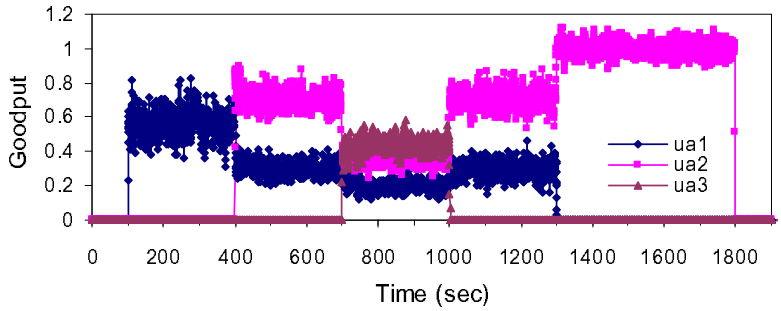

Figure 26: *win-cont* goodput under dynamic load

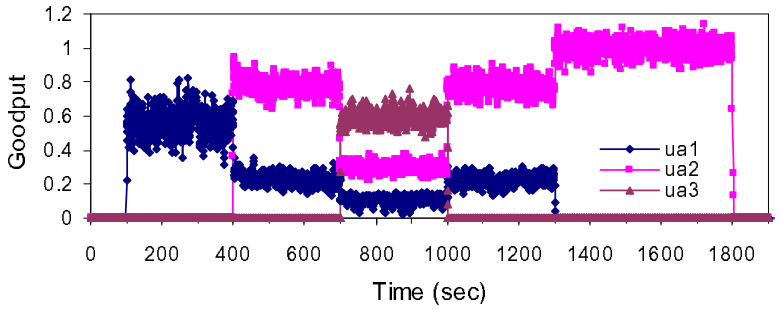

Figure 27: *win-disc* goodput under dynamic load with equal user success rate

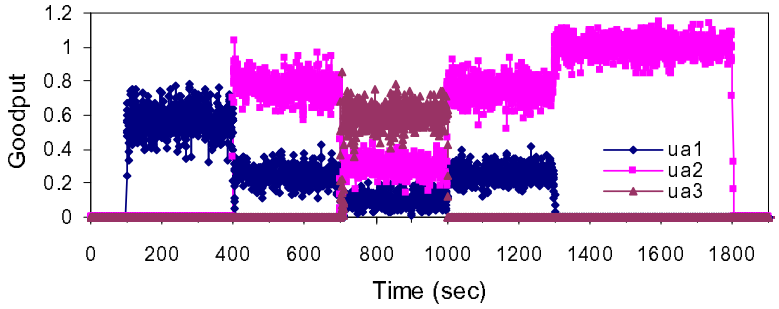

Figure 28: *rate-abs* goodput under dynamic load with equal user success rate

6.2.1 The *rate-abs*, *win-disc* and *win-cont* Control Algorithms

As we mentioned in Section 6.1.2, the inherent property of absolute rate or window feedback mechanisms such as *rate-abs*, *win-disc* and *win-cont* make them impossible to achieve user-centric fairness without additional work. We therefore introduced the double-feed architecture with which the *RE* is informed of the original incoming load from each sources before the load regulation is applied. We then make modifications to the algorithms in Section 3.3, Section 5.3 and Section 5.5. In the *rate-abs* and *win-disc* cases, the modification is straightforward, we simply divide the available total rate or window size proportionally among the active sources. The results are shown in Figure 27 and Figure 28. In both cases the goodput are proportionally shared among the active sources.

In the *win-cont* case, the total available window size during each feedback update could be very small, and most of the times close to one. Simply dividing the small available total window size proportionally and update the feedback as it accumulates to an integer value could cause noticeable differences between the actual share and the expected share of the capacity. This is also the reason why the basic *win-cont* algorithm does not provide equal allocation for the upstream senders. Therefore, we propose an improved *win-cont* allocation algorithm based on weighted fair processing, which gives better accuracy for small window allocation for multiple upstream senders. The algorithm is described as below:

$w_i^0 := W_0$ where $W_0 > 0$
$w_{left'}^0 := 0$
$w_i^t := w_i^t - 1$ for INVITE received from SE_i
$w_{left}^t := N_{sess}^{max} - N_{sess}$ upon msg processing
$w_{left}^t := w_{left}^t + w_{left'}^t$
$if(w_{left}^t \geq 1)$
    $w_{left'}^t = (frac)(w_{left}^t)$

$w^t_{share} = (int)(w^t_{left})$
Assuming the proportion of original load from SE_i is $P\%$
$w^t_i := w^t_{share}$ with probability $P/100$

A simple way to implement the probabilistic allocation of the available window is to assign each active SE a specific range between 0 to 100 . The range length is proportional to the SE's share of the total original load. Each range does not overlap, and the sum of all range length is 100. Each time during an allocation decision, a random number between 0 and 100 is generated. The allocation belongs to the SE whose assigned range covers the generated number. For example, three SEs each contributes 10%, 30%, 60% of the the total incoming load are assigned range value of 0-10, 10-40, and 40-100 respectively. A randomly generated number of 20 falls into the range of te second SE, while a randomly generated number of 50 falls into the range of the third SE. The results from this algorithm is shown in Figure 29 through Figure 31. The goodput from three sources are shown separately for clarity.

It can be seen that UA1 starts at the 100th second with load 0.57 and gets a goodput of the same value. At the 400th second, UA2 is started with load 1.68, three times of UA1's load. UA1's goodput quickly declines and reaches a state where it shares the capacity with UA2 at a one to three proportion. At the 700th second, UA3 is added with a load of 3.36. The combination of the three active sources therefore has a load of 5.6. We see that both UA1 and UA2's goodput immediately decrease. The three sources settle at a stable situation with roughly 0.1, 0.3, and 0.6 goodput, matching the original individual load. At the 1000th second, the bulk arrival of UA3 ends and UA3 left the system. The allocation split between UA1 and UA2 restores to the similar situation before UA3's arrival at the 700th second. Finally, at the 1300th second, UA1 departs the system, leaving UA2 with load 1.68 alone. Since the load is still over the server capacity, UA2 gets exactly the full capacity of the system with a goodput of 1.

Since equal allocation to each sources can be seen as a special case of the proportional allocation, this improved algorithm can naturally achieve the basic service-provider centric fairness for *win-cont* as well.

6.2.2 The *rate-occ* Control Algorithm

The dynamic performance of *rate-occ* is shown in Figure 32 through Figure 34. The goodput change also adapts to the offered load change reasonably fast, although the total goodput does not reach 1 for the same reason as explained in the steady state performance of *rate-occ*. In addition, as can be expected from the original algorithm, *rate-occ* by default achieves user-centric fairness because all the SEs throttle the same percentage of its original incoming load. On the other hand, if *rate-occ* is required to provide basic provider-centric fairness by giving equal allocation to all the SEs, the RE will need to estimate the proportional share of each SE from their incoming load, and then compute the individual throttle percentage for each of them.

6.2.3 The *win-auto* Control Algorithm

Figure 35 through Figure 37 illustrate the dynamic performance of the simplest *win-auto* algorithm. We see that with source arrival and departure, the system still always reaches the maximum goodput as long as the current load is larger than the server capacity. A difference from the other algorithms is that it could take a noticeably longer adaptation time to reach the steady state under certain load surge. For example, if we look at Figure 36 which is the goodput for ua2, at the 700th second when the load increases suddenly from 2.25 to 5.6, it took over 60 s seconds to completely stabilize. However, the good thing is once steady state is reached, the total goodput of all three UAs adds up to one. Moreover, performance under source departure is good. At 1300th second, when UA2 becomes the only UA in the system, its goodput quickly adapts to 1. There is, however, one specific drawback of the *win-auto* mechanism. Since there is basically no processing intervention in this algorithm, we found it hard to enforce an explicit share of the capacity. The outcome of the capacity split seem to be determined by the point

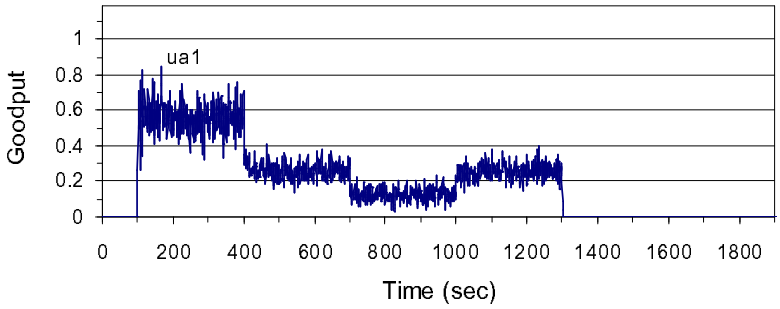

Figure 29: *win-cont* ua1 goodput with dynamic load

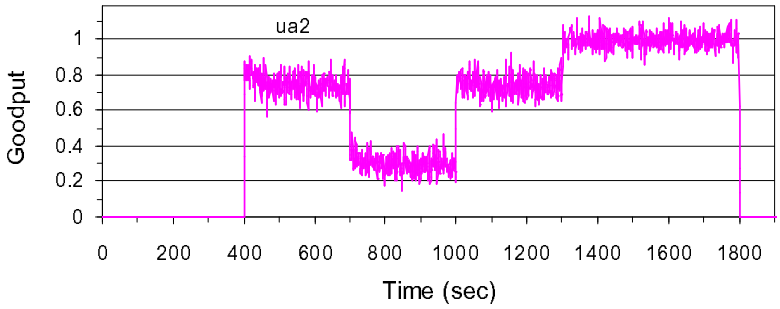

Figure 30: *win-cont* ua2 goodput with dynamic load

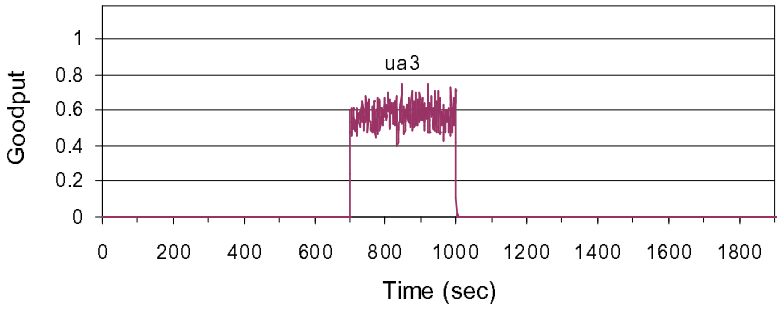

Figure 31: *win-cont* ua3 goodput with dynamic load

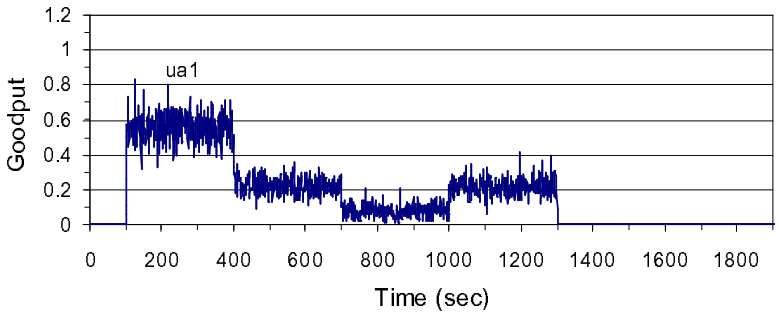

Figure 32: *rate-occ* ua1 goodput with dynamic load

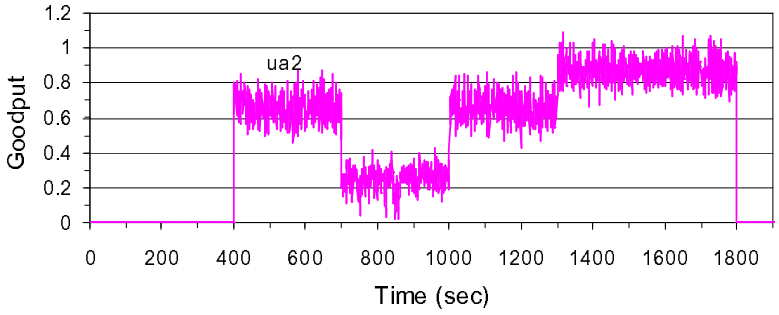

Figure 33: *rate-occ* ua2 goodput with dynamic load

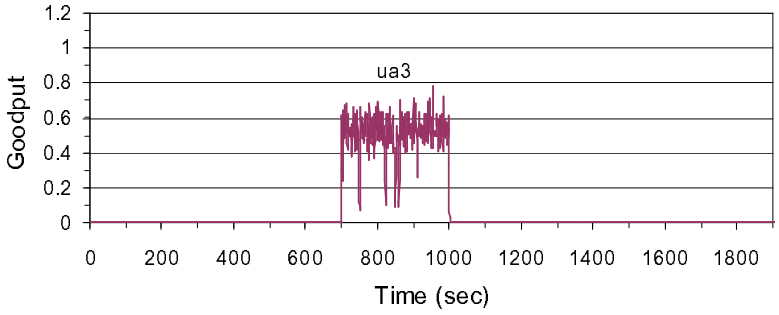

Figure 34: *rate-occ* ua3 goodput with dynamic load

when the system reaching the steady state which is not easy to predict. Therefore, *win-auto* may not be a good candidate when explicit fairness is required. But because of its extreme simplicity, as well as near perfect steady state aggregate performance, *win-auto* may still be a good choice in some situations.

In summary, for dynamic behavior, our simulation shows that all algorithms except *win-auto* adapts well to the offered dynamic load, showing little transition difference during new source arrival and existing source departure as well as at load change boundaries.

As far as fairness is concerned, the *rate-occ* algorithm by default can provide user-centric fairness; the basic *rate-abs*, *win-disc* and *win-cont* algorithms are capable of service provider centric fairness by allocating equal amount of capacity to each SE. After implementing our double-feed architecture with sources reporting the original load to the RE, we are able to achieve user-centric fairness in all *rate-abs*, *win-disc* and *win-cont* algorithms through a proportional allocation of total RE capacity according to SEs' original incoming load.

7 Conclusions and Future Work

The SIP server overload problem is interesting for a number of reasons: first, the cost of rejecting a request could not be ignored as compared to the cost of serving a request; Second, the various SIP timers lead to many retransmissions in overload and amplify the situation; Third, SIP has a server-to-server application level routing architecture. The server-to-server architecture helps the deployment of a pushback SIP overload control solution. The solution can be based on feedback of absolute rate, relative rate, or window size.

We proposed three window adjustment algorithms *win-disc*, *win-cont* and *win-auto* for window-based feedback and resorted to two existing rate adjustment algorithms for absolute rate-based feedback *rate-abs* and relative rate-based feedback *rate-occ*. Among these five algorithms, *win-auto* is the most SIP specific, and *rate-occ* is the least SIP specific. The remaining three *win-disc*, *win-cont*, and *rate-abs* are generic mechanisms, and need to be linked to SIP when being applied to the SIP environment. The common piece that linked them to SIP is the dynamic session estimation algorithm we introduced. It is not difficult to imagine that with the dynamic session estimation algorithm, other generic algorithms can also be applied to SIP.

Now we summarize various aspects of the five algorithms.

The design of most of the feedback algorithms contains a binding parameter. Algorithms binding on queue length or queueing delay such as *win-disc*, *win-cont* and *rate-abs* outperform algorithms binding on processor occupancy such as *rate-occ*. Indeed, all of *win-disc*, *win-cont* and *rate-abs* are able to achieve theoretical maximum performance, meaning the CPU is fully utilized and every message processed contributes to a successful session, with no wasted message in the system at all. On the other hand, occupancy based heuristic is a much coarser control approach. The sensitivity of control also depends on the extra multiplicative increase and decrease parameter tuning. Therefore, from steady load performance and parameter tuning perspective, we favor algorithms other than *rate-occ*.

The adjustment performed by each algorithm can be discrete time driven such as in *win-disc* and *rate-abs*, *rate-occ* or continuous event driven such as in *win-cont* and *win-auto*. Normally the event-driven algorithm could have smaller number of tuning parameters and also be more accurate. But with a sufficiently short discrete time control interval the difference between discrete and continuous adjustments would become small.

We found that all the algorithms except *win-auto* adapts well to traffic source variations as well as bulk arrival overload. When we further look at the fairness property, especially the user-centric fairness which may be preferable in many practical situations, we found the *rate-occ* realizes it by default. All other algorithms except *win-auto* can also achieve it with our introduction of the double-feed SIP overload control architecture.

Finally, *win-auto* frequently needs to be singled out because it is indeed special. With an extremely simple implementation and virtually zero parameter, it archives a remarkable steady load aggregate output in most cases. The tradeoff to this simplicity is a noticeable load adaptation period upon certain load surge, and the difficulty of enforcing explicit fairness models.

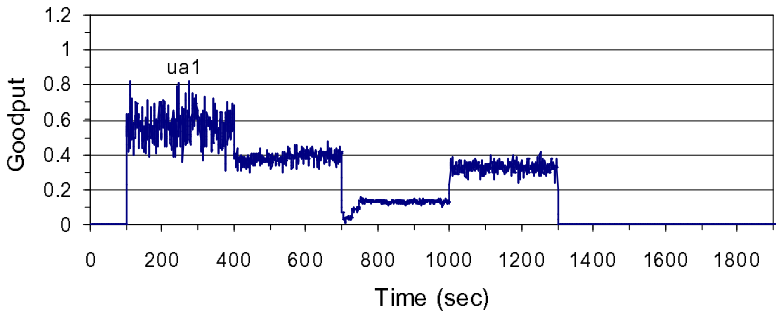

Figure 35: *win-auto* goodput with dynamic load for ua1

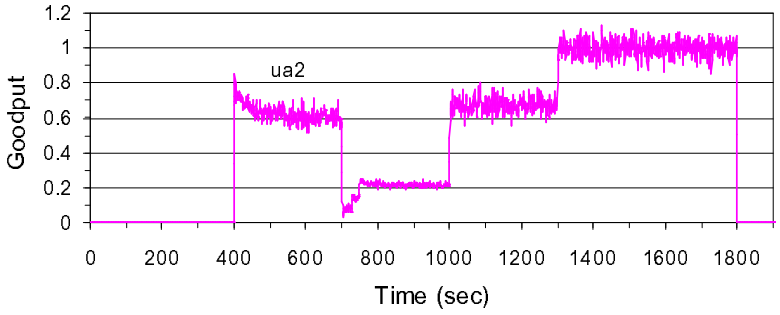

Figure 36: *win-auto* goodput with dynamic load for ua2

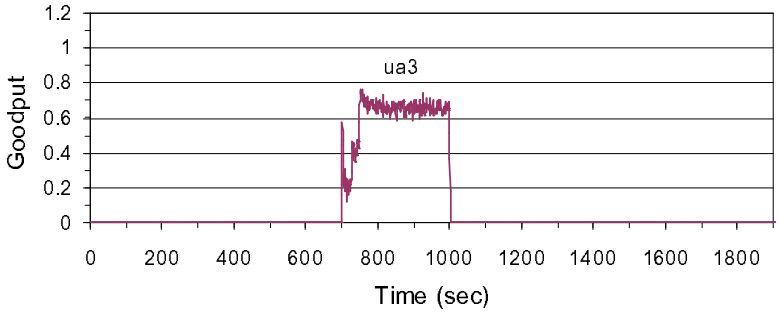

Figure 37: *win-auto* goodput with dynamic load for ua3

Our possible work items for the next step may include adding delay and loss property to the link, and applying other arrival patterns as well as node failure models to make the scenario more realistic. It would be interesting to see whether and how the currently closely matched results of each algorithm may differ in those situations. Another work item is that although we currently assumed percentage-throttle for rate-based and window-throttle for window-based control only, it may be interesting to look at more types of feedback enforcement methods at the SE and see how different feedback algorithms will behave.

8 Acknowledgements

This project receives funding from NTT. The OPNET simulation software is donated by OPNET Technologies under its university program. Part of the research was performed when Charles Shen was an intern at IBM T.J. Watson Research Center. The authors would also like to thank Arata Koike and the IETF SIP overload design team members for helpful discussions.

References

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261 (Proposed Standard), June 2002, Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621.

[2] J. Rosenberg, "Requirements for Management of Overload in the Session Initiation Protocol," Internet draft, Jan. 2008, work in progress.

[3] V. Hilt, I. Widjaja, D. Malas, and H. Schulzrinne, "Session Initiation Protocol (SIP) Overload Control," Internet draft, Feb. 2008, work in progress.

[4] OPNET, "http://www.opnet.com," .

[5] Yoram Bernet, *Networking Quality of Service and Windows Operating Systems*, New Riders, 2001.

[6] P.A. Hosein, "Adaptive rate control based on estimation of message queueing delay," *United States Patent US 6,442,139 B1*, 2002.

[7] Cyr, B.L., Kaufman J.S., and Lee P.T., "Load balancing and overload control in a distributed processing telecommunication systems," *United States Patent US 4,974,256*, 1990.

[8] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, and T. LaPorta, "Fast and robust signaling overload control," *Network Protocols, 2001. Ninth International Conference on*, pp. 323–331, 11-14 Nov. 2001.

[9] M. Ohta, "Overload Protection in a SIP Signaling Network," in *International Conference on Internet Surveillance and Protection (ICISP'06))*, 2006.

[10] Erich Nahum and John Tracey and Charles Wright, "Evaluating SIP server performance," in *ACM SIGMETRICS Performance Evaluation Review*, June 2007, vol. 35, pp. 349–350.

[11] M. Whitehead, "GOCAP - one standardised overload control for next generation networks," *BT Technology Journal*, vol. 23, no. 1, pp. 144–153, 2005.

[12] Eric Noel and Carolyn Johnson, "Initial simulation results that analyze SIP based VoIP networks under overload," in *Proceedings of ITC*, 2007, pp. 54–64.