SIPPING Working Group
Internet-Draft
Intended status: Informational
Expires: January 12, 2010

V. Hilt
Bell Labs/Alcatel-Lucent
E. Noel
AT&T Labs
C. Shen
Columbia University
A. Abdelal
Sonus Networks
July 11, 2009

Design Considerations for Session Initiation Protocol (SIP) Overload Control
draft-ietf-sipping-overload-design-02

Status of this Memo

Copyright Notice

Abstract

Overload occurs in Session Initiation Protocol (SIP) networks when SIP servers have insufficient resources to handle all SIP messages they receive. Even though the SIP protocol provides a limited overload control mechanism through its 503 (Service Unavailable) response code, SIP servers are still vulnerable to overload. This document discusses models and design considerations for a SIP overload control mechanism.

Table of Contents

1. Introduction

As with any network element, a Session Initiation Protocol (SIP) [RFC3261] server can suffer from overload when the number of SIP messages it receives exceeds the number of messages it can process. Overload occurs if a SIP server does not have sufficient resources to process all incoming SIP messages. These resources may include CPU, memory, network bandwidth, input/output, or disk resources.

Overload can pose a serious problem for a network of SIP servers. During periods of overload, the throughput of a network of SIP servers can be significantly degraded. In fact, overload may lead to a situation in which the throughput drops down to a small fraction of the original processing capacity. This is often called congestion collapse.

An overload control mechanism enables a SIP server to perform close to its capacity limit during times of overload. Overload control is used by a SIP server if it is unable to process all SIP requests due to resource constraints. There are other failure cases in which a SIP server can successfully process incoming requests but has to reject them for other reasons. For example, a PSTN gateway that runs out of trunk lines but still has plenty of capacity to process SIP messages should reject incoming INVITEs using a 488 (Not Acceptable Here) response [RFC4412]. Similarly, a SIP registrar that has lost connectivity to its registration database but is still capable of processing SIP messages should reject REGISTER requests with a 500 (Server Error) response [RFC3261]. Overload control mechanisms do not apply in these cases and SIP provides appropriate response codes for them.

The SIP protocol provides a limited mechanism for overload control through its 503 (Service Unavailable) response code and the Retry-After header. However, this mechanism cannot prevent overload of a SIP server and it cannot prevent congestion collapse. In fact, it may cause traffic to oscillate and to shift between SIP servers and thereby worsen an overload condition. A detailed discussion of the SIP overload problem, the problems with the 503 (Service Unavailable) response code and the Retry-After header and the requirements for a SIP overload control mechanism can be found in [RFC5390].

This document discusses the models, assumptions and design considerations for a SIP overload control mechanism. The document is a product of the SIP overload control design team.

2. SIP Overload Problem

A key contributor to the SIP congestion collapse [RFC5390] is the regenerative behavior of overload in the SIP protocol. When SIP is running over the UDP protocol, it will retransmit messages that were dropped by a SIP server due to overload and thereby increase the offered load for the already overloaded server. This increase in load worsens the severity of the overload condition and, in turn, causes more messages to be dropped. A congestion collapse can occur [Noel et al.], [Shen et al.] and [Hilt et al.].

Regenerative behavior under overload should ideally be avoided by any protocol as this would lead to stable operation under overload. However, this is often difficult to achieve in practice. For example, changing the SIP retransmission timer mechanisms can reduce the degree of regeneration during overload but will impact the ability of SIP to recover from message losses. Without any retransmission each message that is dropped due to SIP server overload will eventually lead to a failed call.

For a SIP INVITE transaction to be successful a minimum of three messages need to be forwarded by a SIP server. Often an INVITE transaction consists of five or more SIP messages. If a SIP server under overload randomly discards messages without evaluating them, the chances that all messages belonging to a transaction are successfully forwarded will decrease as the load increases. Thus, the number of transactions that complete successfully will decrease even if the message throughput of a server remains up and assuming the overload behavior is fully non-regenerative. A SIP server might (partially) parse incoming messages to determine if it is a new request or a message belonging to an existing transaction. However, after having spend resources on parsing a SIP message, discarding this message is expensive as the resources already spend are lost. The number of successful transactions will therefore decline with an increase in load as less and less resources can be spent on forwarding messages and more and more resources are consumed by inspecting messages that will eventually be dropped. The slope of the decline depends on the amount of resources spent to inspect each message.

Another challenge for SIP overload control is that the rate of the true traffic source usually cannot be controlled. Overload is often caused by a large number of UAs each of which creates only a single message. These UAs cannot be rate controlled as they only send one message. However, the sum of their traffic can overload a SIP server.

3. Explicit vs. Implicit Overload Control

The main differences between explicit and implicit overload control is the way overload is signaled from a SIP server that is reaching overload condition to its upstream neighbors.

In an explicit overload control mechanism, a SIP server uses an explicit overload signal to indicate that it is reaching its capacity limit. Upstream neighbors receiving this signal can adjust their transmission rate as indicated by the overload signal to a level that is acceptable to the downstream server. The overload signal enables a SIP server to steer the load it is receiving to a rate at which it can perform at maximum capacity.

Implicit overload control uses the absence of responses and packet loss as an indication of overload. A SIP server that is sensing such a condition reduces the load it is forwarding a downstream neighbor. Since there is no explicit overload signal, this mechanism is robust as it does not depend on actions taken by the SIP server running into overload.

The ideas of explicit and implicit overload control are in fact complementary. By considering implicit overload indications a server can avoid overloading an unresponsive downstream neighbor. An explicit overload signal enables a SIP server to actively steer the incoming load to a desired level.

4. System Model

The model shown in Figure 1 identifies fundamental components of an explicit SIP overload control mechanism:

SIP Processor: The SIP Processor processes SIP messages and is the component that is protected by overload control.

Monitor: The Monitor measures the current load of the SIP processor on the receiving entity. It implements the mechanisms needed to determine the current usage of resources relevant for the SIP processor and reports load samples (S) to the Control Function.

Control Function: The Control Function implements the overload control algorithm. The control function uses the load samples (S) and determines if overload has occurred and a throttle (T) needs to be set to adjust the load sent to the SIP processor on the receiving entity. The control function on the receiving entity sends load feedback (F) to the sending entity.

Actuator: The Actuator implements the algorithms needed to act on the throttles (T) and ensures that the amount of traffic forwarded to the receiving entity meets the criteria of the throttle. For example, a throttle may instruct the Actuator to not forward more than 100 INVITE messages per second. The Actuator implements the algorithms to achieve this objective, e.g., using message gapping. It also implements algorithms to select the messages that will be affected and determine whether they are rejected or redirected.

The type of feedback (F) conveyed from the receiving to the sending entity depends on the overload control method used (i.e., loss-based, rate-based, window-based or signal-based overload control; see Section 9), the overload control algorithm (see Section 11) as well as other design parameters. The feedback (F) enables the sending entity to adjust the amount of traffic forwarded to the receiving entity to a level that is acceptable to the receiving entity without causing overload.

Figure 1 depicts a general system model for overload control. In this diagram, one instance of the control function is on the sending entity (i.e., associated with the actuator) and one is on the receiving entity (i.e., associated with the monitor). However, a specific mechanism may not require both elements. In this case, one of two control function elements can be empty and simply passes along feedback. E.g., if (F) is defined as a loss-rate (e.g., reduce traffic by 10%) there is no need for a control function on the sending entity as the content of (F) can be copied directly into (T).

The model in Figure 1 shows a scenario with one sending and one receiving entity. In a more realistic scenario a receiving entity will receive traffic from multiple sending entities and vice versa (see Section 6). The feedback generated by a Monitor will therefore often be distributed across multiple Actuators. A Monitor needs to be able to split the load it can process across multiple sending entities and generate feedback that correctly adjusts the load each sending entity is allowed to send. Similarly, an Actuator needs to be prepared to receive different levels of feedback from different receiving entities and throttle traffic to these entities accordingly.

```
          Sending                 Receiving
           Entity                  Entity
     +----------------+      +----------------+
     |    Server A    |      |    Server B    |
     |  +----------+  |      |  +----------+  |    -+
     |  | Control  |  |  F   |  | Control  |  |     |
     |  | Function |<-+------+--| Function |  |     |
     |  +----------+  |      |  +----------+  |     |
     |     T |        |      |       ^        |     | Overload
     |       v        |      |       | S      |     | Control
     |  +----------+  |      |  +----------+  |     |
     |  | Actuator |  |      |  | Monitor  |  |     |
     |  +----------+  |      |  +----------+  |     |
     |       |        |      |       ^        |    -+
     |       v        |      |       |        |    -+
     |  +----------+  |      |  +----------+  |     |
   <-+--|   SIP    |  |      |  |   SIP    |  |     |  SIP
   --+->|Processor |--+------+->|Processor |--+->    | System
     |  +----------+  |      |  +----------+  |     |
     +----------------+      +----------------+    -+
```

Figure 1: System Model for Explicit Overload Control

5. Degree of Cooperation

A SIP request is usually processed by more than one SIP server on its path to the destination. Thus, a design choice for an explicit overload control mechanism is where to place the components of overload control along the path of a request and, in particular, where to place the Monitor and Actuator. This design choice determines the degree of cooperation between the SIP servers on the path. Overload control can be implemented hop-by-hop with the Monitor on one server and the Actuator on its direct upstream neighbor. Overload control can be implemented end-to-end with Monitors on all SIP servers along the path of a request and an Actuator on the sender. In this case, the Control Functions associated with each Monitor have to cooperate to jointly determine the overall feedback for this path. Finally, overload control can be implemented locally on a SIP server if Monitor and Actuator reside on the same server. In this case, the sending entity and receiving entity are the same SIP server and Actuator and Monitor operate on the same SIP processor (although, the Actuator typically operates on a pre-processing stage in local overload control). Local overload control is an internal overload control mechanism as the control loop is implemented internally on one server. Hop-by-hop and end-to-end are external overload control mechanisms. All three configurations

are shown in Figure 2.

```
               +---------+             +------(+)---------+
      +------+ |         |             |       ^          |
      |      | |        +---+          |       |         +---+
      v      | v    //=>| C |          v       |     //=>| C |
   +---+    +---+ //    +---+       +---+    +---+ //    +---+
   | A |===>| B |                   | A |===>| B |
   +---+    +---+ \\    +---+       +---+    +---+ \\    +---+
               ^    \\=>| D |          ^       |     \\=>| D |
               |        +---+          |       |         +---+
               |         |             |       v          |
               +---------+             +------(+)---------+

         (a) hop-by-hop                   (b) end-to-end

                         +-+
                         v |
    +-+      +-+        +---+
    v |      v |    //=>| C |
   +---+    +---+ //    +---+
   | A |===>| B |
   +---+    +---+ \\    +---+
                    \\=>| D |
                        +---+
                         ^ |
                         +-+

           (c) local

    ==> SIP request flow
    <-- Overload feedback loop
```

Figure 2: Degree of Cooperation between Servers

5.1. Hop-by-Hop

The idea of hop-by-hop overload control is to instantiate a separate control loop between all neighboring SIP servers that directly exchange traffic. I.e., the Actuator is located on the SIP server that is the direct upstream neighbor of the SIP server that has the corresponding Monitor. Each control loop between two servers is completely independent of the control loop between other servers further up- or downstream. In the example in Figure 2(a), three independent overload control loops are instantiated: A - B, B - C and B - D. Each loop only controls a single hop. Overload feedback received from a downstream neighbor is not forwarded further upstream. Instead, a SIP server acts on this feedback, for example, by rejecting SIP messages if needed. If the upstream neighbor of a server also becomes overloaded, it will report this problem to its upstream neighbors, which again take action based on the reported feedback. Thus, in hop-by-hop overload control, overload is always resolved by the direct upstream neighbors of the overloaded server without the need to involve entities that are located multiple SIP hops away.

Hop-by-hop overload control reduces the impact of overload on a SIP network and can avoid congestion collapse. It is simple and scales well to networks with many SIP entities. An advantage is that it does not require feedback to be transmitted across multiple-hops, possibly crossing multiple trust domains. Feedback is sent to the next hop only. Furthermore, it does not require a SIP entity to aggregate a large number of overload status values or keep track of the overload status of SIP servers it is not communicating with.

5.2. End-to-End

End-to-end overload control implements an overload control loop along the entire path of a SIP request, from UAC to UAS. An end-to-end overload control mechanism consolidates overload information from all SIP servers on the way (including all proxies and the UAS) and uses this information to throttle traffic as far upstream as possible. An end-to-end overload control mechanism has to be able to frequently collect the overload status of all servers on the potential path(s) to a destination and combine this data into meaningful overload feedback.

A UA or SIP server only throttles requests if it knows that these requests will eventually be forwarded to an overloaded server. For example, if D is overloaded in Figure 2(b), A should only throttle requests it forwards to B when it knows that they will be forwarded to D. It should not throttle requests that will eventually be forwarded to C, since server C is not overloaded. In many cases, it is difficult for A to determine which requests will be routed to C and D since this depends on the local routing decision made by B. These routing decisions can be highly variable and, for example, depend on call routing policies configured by the user, services invoked on a call, load balancing policies, etc. The fact that a previous message to a target has been routed through an overloaded server does not necessarily mean the next message to this target will also be routed through the same server.

The main problem of end-to-end overload control is its inherent complexity since UAC or SIP servers need to monitor all potential

paths to a destination in order to determine which requests should be throttled and which requests may be sent. Even if this information is available, it is not clear which path a specific request will take.

A variant of end-to-end overload control is to implement a control loop between a set of well-known SIP servers along the path of a SIP request. For example, an overload control loop can be instantiated between a server that only has one downstream neighbor or a set of closely coupled SIP servers. A control loop spanning multiple hops can be used if the sending entity has full knowledge about the SIP servers on the path of a SIP message.

A key difference to transport protocols using end-to-end congestion control such as TCP is that the traffic exchanged between SIP servers consists of many individual SIP messages. Each of these SIP messages has its own source and destination. Even SIP messages containing identical SIP URIs (e.g., a SUBSCRIBE and a INVITE message to the same SIP URI) can be routed to different destinations. This is different from TCP which controls a stream of packets between a single source and a single destination.

5.3. Local Overload Control

The idea of local overload control (see Figure 2(c)) is to run the Monitor and Actuator on the same server. This enables the server to monitor the current resource usage and to reject messages that can't be processed without overusing the local resources. The fundamental assumption behind local overload control is that it is less resource consuming for a server to reject messages than to process them. A server can therefore reject the excess messages it cannot process to stop all retransmissions of these messages.

Local overload control can be used in conjunction with an other overload control mechanisms and provides an additional layer of protection against overload. It is fully implemented within a SIP server and does not require cooperation between servers. In general, SIP servers should apply other overload control techniques to control load before a local overload control mechanism is activated as a mechanism of last resort.

6. Topologies

The following topologies describe four generic SIP server configurations. These topologies illustrate specific challenges for an overload control mechanism. An actual SIP server topology is likely to consist of combinations of these generic scenarios.

In the "load balancer" configuration shown in Figure 3(a) a set of SIP servers (D, E and F) receives traffic from a single source A. A load balancer is a typical example for such a configuration. In this configuration, overload control needs to prevent server A (i.e., the load balancer) from sending too much traffic to any of its downstream neighbors D, E and F. If one of the downstream neighbors becomes overloaded, A can direct traffic to the servers that still have capacity. If one of the servers serves as a backup, it can be activated once one of the primary servers reaches overload.

If A can reliably determine that D, E and F are its only downstream neighbors and all of them are in overload, it may choose to report overload upstream on behalf of D, E and F. However, if the set of downstream neighbors is not fixed or only some of them are in overload then A should not activate an overload control since A can still forward the requests destined to non-overloaded downstream neighbors. These requests would be throttled as well if A would use overload control towards its upstream neighbors.

In the "multiple sources" configuration shown in Figure 3(b), a SIP server D receives traffic from multiple upstream sources A, B and C. Each of these sources can contribute a different amount of traffic, which can vary over time. The set of active upstream neighbors of D can change as servers may become inactive and previously inactive servers may start contributing traffic to D.

If D becomes overloaded, it needs to generate feedback to reduce the amount of traffic it receives from its upstream neighbors. D needs to decide by how much each upstream neighbor should reduce traffic. This decision can require the consideration of the amount of traffic sent by each upstream neighbor and it may need to be re-adjusted as the traffic contributed by each upstream neighbor varies over time.

Server D can use a local fairness policy to determine much traffic it accepts from each upstream neighbor.

In many configurations, SIP servers form a "mesh" as shown in Figure 3(c). Here, multiple upstream servers A, B and C forward traffic to multiple alternative servers D and E. This configuration is a combination of the "load balancer" and "multiple sources" scenario.

```
                   +---+              +---+
                /->| D |              | A |-\
               /   +---+              +---+  \
              /                               \   +---+
       +---+-/     +---+              +---+    \->|   |
       | A |------>| E |              | B |------>| D |
       +---+-\     +---+              +---+    /->|   |
              \                               /   +---+
               \   +---+              +---+  /
                \->| F |              | C |-/
                   +---+              +---+

       (a) load balancer             (b) multiple sources

       +---+
       | A |---\                        a--\
       +---+-\  \---->+---+                 \
              \/----->| D |             b--\ \--->+---+
       +---+--/\  /-->+---+                 \---->|   |
       | B |    \/                      c-------->| D |
       +---+---\/\--->+---+                       |   |
               /\---->| E |            ...   /--->+---+
       +---+--/   /-->+---+                 /
       | C |-----/                      z--/
       +---+
          (c) mesh                      (d) edge proxy
```

Figure 3: Topologies

Overload control that is based on reducing the number of messages a sender is allowed to send is not suited for servers that receive requests from a very large population of senders, each of which only infrequently sends a request. This scenario is shown in Figure 3(d). An edge proxy that is connected to many UAs is a typical example for such a configuration.

Since each UA typically only contributes a few requests, which are often related to the same call, it can't decrease its message rate to resolve the overload. In such a configuration, a SIP server can resort to local overload control by rejecting a percentage of the requests it receives with 503 (Service Unavailable) responses. Since there are many upstream neighbors that contribute to the overall load, sending 503 (Service Unavailable) to a fraction of them can gradually reduce load without entirely stopping all incoming traffic. The Retry-After header can be used in 503 (Service Unavailable) responses to ask UAs to wait a given number of seconds before trying

the call again. Using 503 (Service Unavailable) towards individual sources can, however, not prevent overload if a large number of users places calls at the same time.

> Note: The requirements of the "edge proxy" topology are different than the ones of the other topologies, which may require a different method for overload control.

7. Fairness

There are many different ways to define fairness between multiple upstream neighbors of a SIP server. In the context of SIP server overload, it is helpful to describe two categories of fairness: basic fairness and customized fairness. With basic fairness a SIP server treats all end users equally and ensures that each end user has the same chance of reaching the destination server. With customized fairness, the server allocates resources according to different priorities. An example application of the basic fairness criteria is the "Third caller receives free tickets" scenario, where each end user should have an equal success probability in making calls through an overloaded SIP server, regardless of which service provider he/she is subscribed to. An example of customized fairness would be a server which assigns different resource allocations to its upstream neighbors (e.g., service providers) as defined in a service level agreement (SLA).

8. Performance Metrics

The performance of an overload control mechanism can be measured using different metrics.

A key performance indicator is the goodput of a SIP server under overload. Ideally, a SIP server will be enabled to perform at its capacity limit during periods of overload. E.g., if a SIP server has a processing capacity of 140 INVITE transactions per second then an overload control mechanism should enable it to process 140 INVITEs per second even if the offered load is much higher. The delay introduced by a SIP server is another important indicator. An overload control mechanism should ensure that the delay encountered by a SIP message is not increased significantly during periods of overload.

Reactiveness and stability are other important performance indicators. An overload control mechanism should quickly react to an overload occurrence and ensure that a SIP server does not become overloaded even during sudden peaks of load. Similarly, an overload

control mechanism should quickly remove all throttles if the overload disappears. Stability is another important criteria. An overload control mechanism should not cause significant oscillations of load on a SIP server. The performance of SIP overload control mechanisms is discussed in [Noel et al.], [Shen et al.] and [Hilt et al.].

In addition to the above metrics, there are other indicators that are relevant for the evaluation of an overload control mechanism:

Fairness: Which types of fairness does the overload control mechanism implement?

Self-limiting: Is the overload control self-limiting if a SIP server becomes unresponsive?

Changes in neighbor set: How does the mechanism adapt to a changing set of sending entities?

Data points to monitor: Which and how many data points does an overload control mechanism need to monitor?

9. Explicit Overload Control Feedback

Explicit overload control feedback enables a receiver to indicate how much traffic it wants to receive. Explicit overload control mechanisms can be differentiated based on the type of information conveyed in the overload control feedback and whether the control function is in the receiving or sending entity (receiver- vs. sender-based overload control).

9.1. Rate-based Overload Control

The key idea of rate-based overload control is to limit the request rate at which an upstream element is allowed to forward traffic to the downstream neighbor. If overload occurs, a SIP server instructs each upstream neighbor to send at most X requests per second. Each upstream neighbor can be assigned a different rate cap.

An example algorithm for an Actuator in the sending entity is request gapping. After transmitting a request to a downstream neighbor, a server waits for 1/X seconds before it transmits the next request to the same neighbor. Requests that arrive during the waiting period are not forwarded and are either redirected, rejected or buffered.

The rate cap ensures that the number of requests received by a SIP server never increases beyond the sum of all rate caps granted to upstream neighbors. Rate-based overload control protects a SIP server against overload even during load spikes assuming there are no new upstream neighbors that start sending traffic. New upstream neighbors need to be considered in all rate caps assigned to upstream

neighbors. The overall rate cap of a SIP server is determined by an overload control algorithm, e.g., based on system load.

Rate-based overload control requires a SIP server to assign a rate cap to each of its upstream neighbors while it is activated. Effectively, a server needs to assign a share of its overall capacity to each upstream neighbor. A server needs to ensure that the sum of all rate caps assigned to upstream neighbors is not higher than its actual processing capacity. This requires a SIP server to keep track of the set of upstream neighbors and to adjust the rate cap if a new upstream neighbor appears or an existing neighbor stops transmitting. For example, if the capacity of the server is X and this server is receiving traffic from two upstream neighbors, it can assign a rate of X/2 to each of them. If a third sender appears, the rate for each sender is lowered to X/3. If the overall rate cap is too high, a server may experience overload. If the cap is too low, the upstream neighbors will reject requests even though they could be processed by the server.

An approach for estimating a rate cap for each upstream neighbor is using a fixed proportion of a control variable, X, where X is initially equal to the capacity of the SIP server. The server then increases or decreases X until the workload arrival rate matches the actual server capacity. Usually, this will mean that the sum of the rate caps sent out by the server (=X) exceeds its actual capacity, but enables upstream neighbors who are not generating more than their fair share of the work to be effectively unrestricted. In this approach, the server only has to measure the aggregate arrival rate. However, since the overall rate cap is usually higher than the actual capacity, brief periods of overload may occur.

9.2. Loss-based Overload Control

A loss percentage enables a SIP server to ask an upstream neighbor to reduce the number of requests it would normally forward to this server by a percentage X. For example, a SIP server can ask an upstream neighbor to reduce the number of requests this neighbor would normally send by 10%. The upstream neighbor then redirects or rejects X percent of the traffic that is destined for this server.

An algorithm for the sending entity to implement a loss percentage is to draw a random number between 1 and 100 for each request to be forwarded. The request is not forwarded to the server if the random number is less than or equal to X.

An advantage of loss-based overload control is that, the receiving entity does not need to track the set of upstream neighbors or the request rate it receives from each upstream neighbor. It is sufficient to monitor the overall system utilization. To reduce load, a server can ask its upstream neighbors to lower the traffic forwarded by a certain percentage. The server calculates this percentage by combining the loss percentage that is currently in use (i.e., the loss percentage the upstream neighbors are currently using when forwarding traffic), the current system utilization and the desired system utilization. For example, if the server load approaches 90% and the current loss percentage is set to a 50% traffic reduction, then the server can decide to increase the loss percentage to 55% in order to get to a system utilization of 80%. Similarly, the server can lower the loss percentage if permitted by the system utilization.

Loss-based overload control requires that the throttle percentage is adjusted to the current overall number of requests received by the server. This is particularly important if the number of requests received fluctuates quickly. For example, if a SIP server sets a throttle value of 10% at time t1 and the number of requests increases by 20% between time t1 and t2 (t1<t2), then the server will see an increase in traffic by 10% between time t1 and t2. This is even though all upstream neighbors have reduced traffic by 10% as told. Thus, percentage throttling requires an adjustment of the throttling percentage in response to the traffic received and may not always be able to prevent a server from encountering brief periods of overload in extreme cases.

9.3. Window-based Overload Control

The key idea of window-based overload control is to allow an entity to transmit a certain number of messages before it needs to receive a confirmation for the messages in transit. Each sender maintains an overload window that limits the number of messages that can be in transit without being confirmed.

Each sender maintains an unconfirmed message counter for each downstream neighbor it is communicating with. For each message sent to the downstream neighbor, the counter is increased. For each confirmation received, the counter is decreased. The sender stops transmitting messages to the downstream neighbor when the unconfirmed message counter has reached the current window size.

A crucial parameter for the performance of window-based overload control is the window size. Each sender has an initial window size it uses when first sending a request. This window size can be changed based on the feedback it receives from the receiver.

The sender adjusts its window size as soon as it receives the corresponding feedback from the receiver. If the new window size is

smaller than the current unconfirmed message counter, the sender stops transmitting messages until more messages are confirmed and the current unconfirmed message counter is less than the window size.

Note that the reception of a 100 Trying response does not provide a confirmation for the reception of a message. 100 Trying responses are often created by a SIP server very early in processing and do not indicate that a message has been successfully processed and cleared from the input buffer. If the downstream neighbor is a stateless proxy, it will not create 100 Trying responses at all and instead pass through 100 Trying responses created by the next stateful server. Also, 100 Trying responses are typically only created for INVITE requests. Explicit message confirmations do not have these problems.

Window-based overload control is similar to rate-based overload control in that the total available receiver buffer space needs to be divided among all upstream neighbors. However, unlike rate-based overload control, window-based overload control is self-limiting and can ensure that the receiver buffer does not overflow under normal conditions. The transmission of messages by senders is clocked by message confirmations received from the receiver. A buffer overflow can occur if a large number of new upstream neighbors arrives at the same time. However, senders will eventually stop transmitting new requests once their initial sending window is closed.

In window-based overload control, the number of messages a sender is allowed to send can frequently be set to zero. In this state, the sender needs to be informed when it is allowed to send again and the receiver window has opened up. However, since the sender is not allowed to transmit messages, the receiver cannot convey the new window size by piggybacking it in a response to another message. Instead, it needs to inform the sender through another mechanism, e.g., by sending a message that contains the new window size.

9.4. Overload Signal-based Overload Control

The key idea of overload signal-based overload control is to use the transmission of a 503 (Service Unavailable) response as a signal for overload in the downstream neighbor. After receiving a 503 (Service

Unavailable) response, the sender reduces the load forwarded to the downstream neighbor to avoid triggering more 503 (Service Unavailable) responses. The sender keeps reducing the load if more 503 (Service Unavailable) responses are received. Note that this scheme is based on the use of 503 (Service Unavailable) responses without Retry-After header as the Retry-After header would require a sender to entirely stop forwarding requests.

A sender which has not received 503 (Service Unavailable) responses for a while but is still throttling traffic can start to increase the offered load. By slowly increasing the traffic forwarded a sender can detect that overload in the downstream neighbor has been resolved and more load can be forwarded. The load is increased until the sender again receives another 503 (Service Unavailable) response or is forwarding all requests it has. A possible algorithm for adjusting traffic is additive increase/multiplicative decrease (AIMD).

Overload Signal-based Overload Control is a sender-based overload control mechanism.

9.5. On-/Off Overload Control

On-/off overload control feedback enables a SIP server to turn the traffic it is receiving either on or off. The 503 (Service Unavailable) response with Retry-After header implements on-/off overload control. On-/off overload control is less effective in controlling load than the fine grained control methods above. In fact, all above methods can realize on/-off overload control, e.g., by setting the allowed rate to either zero or unlimited.

10. Implicit Overload Control

Implicit overload control ensures that the transmission of a SIP server is self-limiting. It slows down the transmission rate of a sender when there is an indication that the receiving entity is experiencing overload. Such an indication can be that the receiving entity is not responding within the expected timeframe or is not responding at all. The idea of implicit overload control is that senders should try to sense overload of a downstream neighbor even if there is no explicit overload control feedback. It avoids that an overloaded server, which has become unable to generate overload control feedback, will be overwhelmed with requests.

Window-based overload control is inherently self-limiting since a sender cannot continue without receiving confirmations. All other explicit overload control schemes described above do not have this property and require additional implicit controls to limit transmissions in case an overloaded downstream neighbor does not generate explicit feedback.

11. Overload Control Algorithms

An important aspect of the design of an overload control mechanism is

the overload control algorithm. The control algorithm determines when the amount of traffic to a SIP server needs to be decreased and when it can be increased. In terms of the model described in Section 4 the control algorithm takes (S) as an input value and generates (T) as a result.

Overload control algorithms have been studied to a large extent and many different overload control algorithms exist. With many different overload control algorithms available, it seems reasonable to suggest a baseline algorithm in a specification for a SIP overload control mechanism and allow the use of other algorithms if they provide the same protocol semantics. This will also allow the development of future algorithms, which may lead to a better performance.

12. Message Prioritization

Overload control can require a SIP server to prioritize messages and select messages that need to be rejected or redirected. The selection is largely a matter of local policy of the SIP server. As a general rule, a SIP server should preserve high-priority requests such as emergency service requests as much as possible during times of overload. It should also prioritize messages for ongoing sessions over messages that set up a new session.

13. Security Considerations

Overload control mechanisms, in general, have security implications. If not designed carefully they can, for example, be used to launch a denial of service attack. The specific security risks and their remedies depend on the actual protocol mechanisms chosen for overload control. They need to be addressed in a document that specifies such a mechanism.

14. IANA Considerations

This document does not require any IANA considerations.

15. Informative References

[Hilt et al.]
Hilt, V. and I. Widjaja, "Controlling Overload in Networks of SIP Servers", IEEE International Conference on Network Protocols (ICNP'08), Orlando, Florida, October 2008.

[Noel et al.]
Noel, E. and C. Johnson, "Initial Simulation Results That Analyze SIP Based VoIP Networks Under Overload", International Teletraffic Congress (ITC'07), Ottawa, Canada, June 2007.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC4412] Schulzrinne, H. and J. Polk, "Communications Resource Priority for the Session Initiation Protocol (SIP)", RFC 4412, February 2006.

[RFC5390] Rosenberg, J., "Requirements for Management of Overload in the Session Initiation Protocol", RFC 5390, December 2008.

[Shen et al.]

Shen, C., Schulzrinne, H., and E. Nahum, "Session Initiation Protocol (SIP) Server Overload Control: Design and Evaluation, Principles", Systems and Applications of IP Telecommunications (IPTComm'08), Heidelberg, Germany, July 2008.

Appendix A. Contributors

Contributors to this document are: Mary Barnes (Nortel), Carolyn Johnson (AT&T Labs), Daryl Malas (CableLabs), Tom Phelan (Sonus Networks), Jonathan Rosenberg (Cisco), Henning Schulzrinne (Columbia University), Nick Stewart (British Telecommunications plc), Rich Terpstra (Level 3), Fangzhe Chang (Bell Labs/Alcatel-Lucent). Many thanks!

Authors' Addresses

Volker Hilt
Bell Labs/Alcatel-Lucent
791 Holmdel-Keyport Rd
Holmdel, NJ 07733
USA

Email: volkerh@bell-labs.com

Eric Noel
AT&T Labs

Email: eric.noel@att.com

Charles Shen
Columbia University

Email: charles@cs.columbia.edu

Ahmed Abdelal
Sonus Networks

Email: aabdelal@sonusnet.com