Requirements for Management of Overload in the Session Initiation Protocol
draft-rosenberg-sipping-overload-reqs-02

Status of this Memo

Copyright Notice

Abstract

Overload occurs in Session Initiation Protocol (SIP) networks when proxies and user agents have insuffient resources to complete the processing of a request. SIP provides limited support for overload handling through its 503 response code, which tells an upstream element that it is overloaded. However, numerous problems have been identified with this mechanism. This draft summarizes the problems with the existing 503 mechanism, and provides some requirements for a solution.

Table of Contents

1. Introduction

Overload occurs in Session Initiation Protocol (SIP) [1] networks when proxies and user agencies have insuffient resources to complete the processing of a request or a response. SIP provides limited support for overload handling through its 503 response code, which tells an upstream element that it is overloaded. However, numerous problems have been identified with this mechanism.

This draft describes the general problem of SIP overload, and then reviews the current SIP mechanisms for dealing with overload. It then explains some of the problems with these mechanisms. Finally, the document provides a set of requirements for fixing these problems.

2. Causes of Overload

Overload occurs when an element, such as a SIP user agent or proxy, has insufficient resources to keep up with the volume of traffic it is receiving. Resources include all of the capabilities of the element used to process a request, including CPU processing, memory, I/O, or disk resources. It can also include external resources, such as a database or DNS server. Overload can occur for many reasons, including:

Poor Capacity Planning: SIP networks need to be designed with sufficient numbers of servers, hardware, disks, and so on, in order to meet the needs of the subscribers they are expected to serve. Capacity planning is the process of determining these needs. It is based on the number of expected subscribers and the types of flows they are expected to use. If this work is not done properly, the network may have insufficient capacity to handle predictable usages, including regular usages and predictably high ones (such as high voice calling volumes on Mothers Day).

Dependency Failures: A SIP element can become overloaded because a resource on which it is dependent has failed, greatly reducing its actual capacity. As such, even minimal traffic might cause the server to go into overload. Examples of such dependency failures include DNS servers, databases, disks and network interfaces.

Component Failures: A SIP element can become overloaded when it is a member of a cluster of servers which each share the load of traffic, and one or more of the other members in the cluster fail. In this case, the remaining elements take over the work of the failed elements. Normally, capacity planning takes such failures into account, and servers are typically run with enough spare

capacity to handle failure of another element. However, unusual failure conditions can cause many elements to fail at once. This is often the case with software failures, where a bad packet or bad database entry hits the same bug in a set of elements in a cluster.

Avalanche Restart: One of the most troubling sources of overload is avalanche restart. This happens when a large number of clients all simultaneously attempt to connect to the network with a SIP registration. Avalanche restart can be caused by several events. One is the "Manhattan Reboots" scenario, where there is a power failure in a large metropolitan area, such as Manhattan. When power is restored, all of the SIP phones, whether in PCs or standalone devices, simultaneously power on and begin booting. They will all then connect to the network and register, causing a flood of SIP REGISTER messages. Another cause of avalanche restart is failure of a large network connection, for example, the access router for an enterprise. When it fails, SIP clients will detect the failure rapidly using the mechanisms in [4]. When connectivity is restored, this is detected, and clients re-REGISTER, all within a short time period. Another source of avalanche restart is failure of a proxy server. If clients had all connected to the server with TCP, its failure will be detected, followed by re-connection and re-registration to another server. Note that [4] does provide some remedies to this case.

Flash Crowds: A flash crowd occurs when an extremely large number of users all attempt to simultaneously make a call. One example of how this can happen is a television commercial that advertises a number to call to receive a free gift. If the gift is compelling and many people see the ad, many calls can be simultaneously made to the same number. This can send the system into overload.

Unfortunately, the overload problem tends to compound itself. When a network goes into overload, this can frequently cause failures of the elements that are trying to process the traffic. This causes even more load on the remaining elements. Furthermore, during load, the overall capacity of functional elements goes down, since much of their resources are spent just rejecting or treating load that they cannot actually process. In addition, overload tends to cause SIP messages to delayed or be lost, which causes retransmissions to be sent, further increasing the amount of work in the network. This compounding factor can produce substantial multipliers on the load in the system. Indeed, with as many as 7 retransmits of an INVITE request prior to timeout, overload can multiply the already-heavy message volume by as much as seven!

3. Current SIP Mechanisms

SIP provides very basic support for overload. It defines the 503 response code, which is sent by an element that is overloaded. RFC 3261 defines it thusly:

> The server is temporarily unable to process the request due to a temporary overloading or maintenance of the server. The server MAY indicate when the client should retry the request in a Retry-After header field. If no Retry-After is given, the client MUST act as if it had received a 500 (Server Internal Error) response.
>
> A client (proxy or UAC) receiving a 503 (Service Unavailable) SHOULD attempt to forward the request to an alternate server. It SHOULD NOT forward any other requests to that server for the duration specified in the Retry-After header field, if present.
>
> Servers MAY refuse the connection or drop the request instead of responding with 503 (Service Unavailable).

The objective is to provide a mechanism to move the work of the overloaded server to another server, so that the request can be processed. The Retry-After header field, when present, is meant to allow a server to tell an upstream element to back off for a period of time, so that the overloaded server can work through its backlog of work.

RFC3261 also instructs proxies to not forward 503 responses upstream, at SHOULD NOT strength. This is to avoid the upstream server of mistakingly concluding that the proxy is overloaded, when in fact the problem was an element further downstream.

4. Problems with the Mechanism

At the surface, the 503 mechanism seems workable. Unfortunately, this mechanism has had numerous problems in actual deployment. These problems are described here.

4.1. Load Amplification

The principal problem with the 503 mechanism is that it tends to substantially amplify the load in the network when the network is overloaded, causing further escalation of the problem and introducing the very real possibility of congestive collapse. Consider the topology in Figure 2.

```
                                          +------+
                                        > |      |
                                       /  |  S1  |
                                      /   |      |
                                     /    +------+
                                    /
                                   /
                                  /
                                 /
                     +------+ /           +------+
            --------> |      |/            |      |
                      |  P1  |---------> |  S2  |
            --------> |      |\            |      |
                     +------+ \           +------+
                                 \
                                  \
                                   \
                                    \
                                     \
                                      \   +------+
                                       \  |      |
                                        > |  S3  |
                                          |      |
                                          +------+
```

Figure 2

Proxy P1 receives SIP requests from many sources, and acts solely as a load balancer, proxying the requests to servers S1, S2 and S3 for processing. The input load increases to the point where all three servers become overloaded. Server S1, when it receives its next request, generates a 503. However, because the server is loaded, it might take some time to generate the 503, causing request retransmissions which further increase the work on S1. When the 503 is received by P1, it retries the request on S2. S2 is also overloaded, and eventually generates a 503, but in the interim is also hit with many retransmits. P1 once again tries another server, this time S3, which also eventually rejects it with a, but only after many retransmits of the request.

Thus, the processing of this request, which ultimately failed, involved four SIP transactions, each of which involved many retransmissions - up to 7. Thus, under unloaded conditions, a single request from a client would generate one request (to S1, S2 or S3) and two responses. How, a single request from the client, before timing out, could generate as many as 18 requests and as many responses! Each server had to expend resources to process these message. Thus, more messages and more work were sent into the

network at the point at which the elements became overloaded. The 503 mechanism works well when a single element is overloaded. But, when the problem is overall network load, the 503 mechanism actually generates more messages and more work for all servers, ultimately resulting in the rejection of the request anyway.

The problem becomes amplified further if one considers proxies upstream from P1, as shown in Figure 3.

```
                                +------+
                              > |      | <
                             /  |  S1  |  \\
                            /   |      |    \\
                           /    +------+      \\
                          /                     \
                         /                       \\
                        /                          \\
                       /                             \
            +------+  /         +------+           +------+
            |      | /          |      |           |      |
            |  P1  | ---------> |  S2  |<----------|  P2  |
            |      | \          |      |           |      |
            +------+  \         +------+           +------+
                ^      \                             / ^
                 \      \                          // /
                  \      \                       //  /
                   \      \                    //   /
                    \      \                  /    /
                     \      \   +------+    //    /
                      \      \  |      |  //     /
                       \      > |  S3  | <      /
                        \       |      |       /
                         \      +------+      /
                          \                  /
                           \                /
                            \              /
                             \            /
                              \          /
                               \        /
                                \      /
                                 \    /
                                +------+
                                |      |
                                |  PA  |
                                |      |
                                +------+
                                 ^   ^
                                 |   |
                                 |   |
```

Figure 3

Here, proxy PA receives requests, and sends these to proxies P1 or P2. P1 and P2 both load balance across S1 through S3. Assuming again S1 through S3 are all overloaded, a request arrives at PA, which tries P1 first. P1 tries S1, S2 and then S3, and each transaction resulting in many request retransmits. Since P1 is

unable to eventually process the request, it rejects it. However, since all of its downstream dependencies are busy, it decides to send a 503. This propagates to PA, which tries P2, which tries S1 through S3 again, resulting in a 503 once more. Thus, in this case, we have doubled the number of SIP transactions and overall work in the network compared to the previous case.

4.2. Underutilization

Interestingly, there are also examples of deployments where the network capacity was greatly reduced as a consequence of the overload mechanism. Consider again Figure 2. Unfortunately, RFC 3261 is unclear on the scope of a 503. When it is received by P1, does the proxy cease sending requests to that IP address? To the hostname? To the URI? Some implementations have chosen the hostname as the scope. When the hostname for a URI points to an SRV record in the DNS, which, in turn, maps to a cluster of downstream servers (S1, S2 and S3 in the example), a 503 response from a single one of them will make the proxy believe that the entire cluster is overloaded. Consequently, proxy P1 will cease sending any traffic to any element in the cluster, even though there are elements in the cluster that are underutilized.

4.3. The Off/On Retry-After Problem

The Retry-After mechanism allows a server to tell an upstream element to stop sending traffic for a period of time. The work that would have otherwise been sent to that server is instead sent to another server. The mechanism is an all-or-nothing technique. A server can turn of all traffic towards it, or none of it. There is nothing in between. This tends to cause highly oscillatory behavior under even mild overload. Consider a proxy P1 which is balancing requests between two servers S1 and S2. The input load just reaches the point where both S1 and S2 are at 100% capacity. A request arrives at P1, and is sent to S1. S1 rejects this request with a 503 , and decides to use Retry-After to clear its backlog. P1 stops sending all traffic to S1. Now, S2 gets traffic, but it is seriously overloaded - at 200% capacity! It decides to reject a request with a 503 and a Retry-After, which now forces P1 to reject all traffic until S1's Retry-After timer expires. At that point, all load is shunted back to S1, which reaches overload, and the cycle repeats.

Its important to observe that this problem is only observed for servers where there are a small number of upstream elements sending it traffic, as is the case in these examples. If a proxy was accessed by a large number of clients, each of which sends a small amount of traffic, the 503 mechanism with Retry-After is quite effective when utilized with a subset of the clients. This is

because spreading the 503 out amongst the clients has the effect of providing the proxy more fine-grained controls on the amount of work it receives.

4.4. Ambiguous Usages

Unfortunately, the specific instances under which a server is to send a 503 are ambiguous. The result is that implementations generate 503 for many reasons, only some of which are related to actual overload. For example, RFC 3398 [2], which specifies interworking from SIP to ISUP, defines the usage of 503 when the gateway receives certain ISUP cause codes from downstream switches. In these cases, the gateway has ample capacity; its just that this specific request could not be processed because of a downstream problem.

This causes two problems. Firstly, during periods of overload, it exacerbates the problems above because it causes additional 503 to be fed into the system, causing further work to be generated in conditions of overload. The other problem is that it becomes hard for an upstream element to know whether to retry when a 503 is received. There are classes of failures where trying on another server won't help, since the reason for the failure was that a common downstream resource is unavailable. For example, if servers S1 and S2 share a database, and the database fails. A request sent to S1 will result in a 503, but retrying on S2 won't help since the same database is unavailable.

5. Solution Requirements

In this section, we propose requirements for an overload control mechanism for SIP which addresses these problems.

REQ 1: The overload mechanism shall strive to maintain the overall useful throughput (taking into consideration the quality-of-service needs of the using applications) of a SIP at reasonable levels even when the incoming load on the network is far in excess of its capacity. The overall throughput under load is the ultimate measure of the value of an overload control mechanism.

REQ 2: When a single network element fails, goes into overload, or suffers from reduced processing capacity (possibly due to unavailability of other resources, such as databases or DNS), the mechanism should strive to limit the impact of this on other elements in the network. This helps to prevent a small-scale failure from becoming a widespread outage.

REQ 3: The mechanism should seek to minimize the amount of configuration required in order to work. For example, it is better to avoid needing to configure a server with its SIP message throughput, as these kinds of quantities are hard to determine.

REQ 4: The mechanism must be capable of dealing with elements which do not support it, so that a network can consist of a mix of ones which do and don't support it. In other words, the mechanism should not work only in environments where all elements support it. It is reasonable to assume that it works better in such environments, of course. Ideally, there should be incremental improvements in overall network throughput as increasing numbers of elements in the network support the mechanism.

REQ 5: The mechanism should not assume that it will only be deployed in environments with completely trusted elements. It should seek to operate as effectively as possible in environments where other elements are malicious, including preventing malicious elements from obtaining more than a fair share of service.

REQ 6: The mechanism shall provide a way to unambiguously inform an upstream element that it is overloaded. Any response codes, header fields, or other protocol machinery utilized for this purpose shall be used exclusively for overload handling, and not be used to indicate other failure conditions. This is meant to avoid some of the problems that have arisen from the reuse of the 503 response code for multiple purposes.

REQ 7: The mechanism shall provide a way for an element to throttle the amount of traffic it receives from an upstream element. This throttling shall be graded, so that it is not all or nothing as with the current 503 mechanism. This recognizes the fact that "overload" is not a binary state, and there are degrees of overload.

REQ 8: The mechanism shall ensure that, when a request has been rejected from an overloaded element, it is not sent to another element suffering from greater levels of load. This requirement derives from REQ 1.

REQ 9: That a request has been rejected from an overloaded element shall not unduly restrict the ability of that request to be submitted to and processed by an element that is less overloaded. This requirement derives from REQ 1.

REQ 10: The mechanism should support servers that receive requests from a large number of different upstream elements, where the set of upstream elements is not enumerable.

REQ 11: The mechanism should support servers that receive requests from a finite set of upstream elements, where the set of upstream elements is enumerable.

REQ 12: The mechanism should work between servers in different domains.

REQ 13: The mechanism must not dictate a specific algorithm for prioritizing the processing of work within a proxy during times of overload. It must permit a proxy to prioritize requests based on any local policy, so that certain ones (such as a call for emergency services or a call with a specific value of of the Resource-Priority header field [3]) are processed ahead of others.

REQ 14: The mechanism should provide unambigous directions to clients on when they should retry a request, and when they should not. This especially applies to TCP connection establishment and SIP registrations, in order to mitigate against avalanche restart.

REQ 15: In cases where a network element fails, is so overloaded that it cannot process messages, or cannot communicate due to a network failure or network partition, it will not be able to provide explicit indications of its levels of congestion. The mechanism should properly function in these cases.

REQ 16: The mechanism should attempt to minimize the overhead of the overload control messaging.

REQ 17: The overload mechanism must not provide an avenue for malicious attack.

REQ 18: The overload mechanism should be unambiguous about whether a load indication applies to a specific IP address, host, or URI, so that an upstream element can determine the load of the entity to which a request is to be sent.

REQ 19: The specification for the overload mechanism should give guidance on which message types might be desirable to process over others during times of overload, based on SIP-specific considerations. For example, it may be more beneficial to process a SUBSCRIBE refresh with Expires of zero than a SUBSCRIBE refresh with a non-zero expiration, since the former reduces the overall amount of load on the element, or to process re-INVITEs over new INVITEs.

REQ 20: In a mixed environment of elements that do and do not implement the overload mechanism, no disproportionate benefit shall accrue to the users or operators of the elements that do not implement the mechanism.

6. Simulation Model

In order to analyze the problem and compare solutions, it is useful to have a baseline simulation model that can be used. This section defines such a model. It is broken up into a model of the network, a model of the user agents, a model of a proxy, and a set of ranges and proposed defaults for the simulation parameters.

6.1. Modeling the Network

```
                      +-----------+    +-----------+
                      |           |    |           |
                      |   Home    |    |   Home    |
                      |  Proxy    |    |  Proxy    |
                      |           |    |           |
                      +-----------+    +-----------+
                        /      \        /     \
                       /        \      /       \
                      /          \    /         \
              +-----------+   +-----------+   +-----------+
              |           |   |           |   |           |
              |   Edge    |   |   Edge    |   |   Edge    |
              |  Proxy    |   |  Proxy    |   |  Proxy    |
              |           |   |           |   |           |
              +-----------+   +-----------+   +-----------+
                    /            /   \             \
                   /            /     \             \
                  /            /       \             \
                 /            /         \             \
                /            /           \             \
          +--------+    +--------+    +--------+    +--------+
          |        |    |        |    |        |    |        |
          |   UA   |    |   UA   |    |   UA   |    |   UA   |
          |        |    |        |    |        |    |        |
          +--------+    +--------+    +--------+    +--------+
```

Figure 4

Figure 4 depicts a network diagram for the purposes of simulation. There are a large number of user agents in the system (Nua). There are a smaller number of edge proxies (Nep), which sit between the UA

and the rest of the SIP network. The user agents send SIP requests towards the edge proxies, which perform functions such as SIP compression and authentication, and then forward them towards the home proxies. There are fewer home proxies (Nhp). These proxies process the request, including functions such as authorization, accounting, and call routing. They then forward requests back towards one of the edge proxies, which in turn deliver the request to a UA.

For purposes of simulation, it is assumed that each UA is associated with two of the edge proxies, randomly selected amongst the set of Nep edge proxies. The UA will send all of its requests towards one of the two unless that one has failed, in which case it sends its traffic to the other one. Each edge proxy forwards requests it receives from the UA to one of the Nhp home proxies. We assume the requests are distributed uniformly amongst the proxies. Similarly, messages sent from the home proxy to the edge proxies are distributed uniformly amongst them. For purposes of simulation, the edge proxy delivers a request received from an edge proxy to one of the user agents arbitrarily.

It is assumed that there is a single network between the UA and the edge proxies, and one between the edge proxies and the home proxy. Each network is modeled as a queue. When an element sends a request, it is enqueued, or dropped if the queue is full. The queue is serviced with at a fixed bandwidth. A packet is delivered to the recipient once the packet could have been completely sent, based on its size and the service rate. The service rate on the network between the UAs and edge proxies is serviced at a rate of Baccess bits per second, and between the edge proxies and home proxy, at Bcore bits per second. The size of the buffers are Saccess and Score for the UA to edge and edge to core networks, respectively.

In addition, when a packet is enqueued in the access network, there is a Placcess probability that it is immediately discarded. In the core network, this probability is Plcore. This models packet loss due to other factors besides the presence of the SIP traffic being modeled with the queue.

Though the network model is simple, and more complex models including different queueing and service disciplines is possible, the impact of the network on the system is a secondary phenomenon and thus a detailed model is not required.

6.2. Modeling the User Agents

Each user agent initiates SIP transactions based on a poisson distribution with arrival rate Rnew. The model considers only the

"busy hour" and consequently a high value for Rnew (discussed below) is used. The transaction can either be an INVITE transaction or a non-INVITE transaction. Whether it is INVITE or non-INVITE is a boolean variability with probability of INVITE being equal to Pinv. Consequently the arrival rate of INVITE transactions from one client is Poisson with arrival rate Rnew*Pinv.

When a transaction is initiated, the request is sent using UDP. This requires the client to retransmit the request and process responses based on the state machine in Section 17.1 of RFC 3261. Each UDP packet, whether request or response, is assumed to be Spkt bytes in size. It is assumed that each UA has infinite processing capacity, and can therefore instantly send a request when required by the state machine, or process a response instantly when one is received. The model does not try to capture overload of the end points themselves.

The model does not try to more accurately capture network traffic loads through means of standardized call setup and hold times, registration times and so on. Though useful, the impact of this is also considered to be secondary on the overload processing, which is more strongly coupled to the mix of transaction types and overall load.

6.3. Modeling the Proxies

```
                                    |
                                    |
                                    |
            +-----------------------|-------------------------+
            |                       |                         |
            |                       V                         |
            |                     |    |                      |
            |                     |----|                      |
            |                     |----|                      |
            |                     |----|                      |
            |                     +----+                      |
            |                        |                        |
            |                        V                        |
            |                 +-------------+                 |
            |                 |             |                 |
            |                 |             |                 |
            |                 |  Parse and  |                 |
            |                 |  PreProcess |                 |
            |                 |             |                 |
            |                 |             |                 |
            |                 +-------------+                 |
            |                        |                        |
            |                        V                        |
            |              |    |  |    |  |    |             |
            |              |----|  |----|  |----|             |
            |              |----|  |----|  |----|             |
            |              |----|  |----|  |----|             |
            |              +----+  +----+  +----+             |
            |                        |                        |
            |                        V                        |
            |                 +-------------+                 |
            |                 |             |                 |
            |                 |             |                 |
            |                 |   Process   |                 |
            |                 |             |                 |
            |                 |             |                 |
            |                 |             |                 |
            |                 +-------------+                 |
            |                        |                        |
            |                        |                        |
            +------------------------|------------------------+
                                     |
                                     |
                                     V
```

Figure 5

A model for a proxy server is shown in Figure 5. Packets, whether they are requests or responses, arrive at the top and are enqueued. The queue is of depth Spin bits. If the queue is full the incoming packet is discarded. The queue is serviced by a component which performs pre-processing and parsing on the message. This pre-processing will determine the type of the message and determine a classification used for enqueueing in a second queue. This allows the model to accommodate prioritization algorithms which might prefer responses over requests, or high priority requests over normal ones. The proxy is modeled as having a fixed capacity of Ch units/s for the home proxy and Ce for the edge proxy. This cost models the overall CPU capacity which can be spread across the various tasks in the system. To choose a useful normalization for the value, the cost of processing an INVITE request is modeled as one unit. The pre-processing component can service requests at a cost of Cpreq units per request, and responses at a cost of Cpres units per response. It is also capable of rejecting requests in cases of overload, at a cost of Cprej units per request. A request will get rejected if the second level queue is full.

There can be one or more second level queues, each for a different type of message which is to be handled separately. In the simplest case there is only one such queue. The depth of each queue is Srin. All of these queues are serviced by a processing component. When processing a request, this component implements the server transaction described in Section 17.2 of RFC 3261, followed by the client side transaction in Section 17.1 of RFC 3261. When processing a response, this component implements the client transaction in Section 17.1 of RFC 3261 followed by the server transaction in Section 17.2 of RFC 3261. This model assumes there is no forking; a request is delivered to a single next-hop destination as described above.

The processing component can process requests at a cost of Ris units per INVITE request, Rnis units per non-INVITE request, Rirs units per INVITE response, and Rnirs units per non-INVITE response.

6.4. Model Parameter Values

The table below enumerates the parameters of the model, gives typical ranges, and suggests a default value.

Parameter Name	Unit	Range	Default
Nua	none	10e3-50e6	100e3
Nep	none	1-100	4
Nhp	none	1-50	2
Baccess	bits/s	100e6-100e9	100e6
Bcore	bits/s	100e6-100e9	1e9
Saccess	bits	1e3 - 1e6	2e3
Score	bits	1e3 - 1e6	2e3
Placcess	none	0-1	.02
Plcore	none	0-1	0
Pinv	none	0-1	.4
Spkit	bytes	1e2-10e3	8e2
Rnew	1/hour	.1 - 10	4
Spin	bits	1e3-1e6	2e3
Ch	units	10-10e3	500
Ce	unites	10-10e3	500
Cpreq	units	1e-3 - 1	1e-2
Cpres	units	1e-3 - 1	1e-2
Cprej	units	1e-3 - 1	8e-2
Srin	bits	1e3-1e6	2e3
Ris	units	1	1
Rnis	units	1e-2 to 1e1	1e-1
Rirs	units	1e-4 to 1	1e-2
Rnirs	units	1e-4 to 1	1e-2

Figure 6

7. Security Considerations

Like all protocol mechanisms, a solution for overload handling must prevent against malicious inside and outside attacks. This document includes requirements for such security functions.

8. IANA Considerations

None.

9. Acknowledgements

The author would like to thank Steve Mayer, Mouli Chandramouli, Robert Whent, Mark Perkins, Joe Stone, Vijay Gurbani, Steve Norreys, and Dale Worley for their contributions to this document.

10. Informative References

[1] 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.

[2] Camarillo, G., Roach, A., Peterson, J., and L. Ong, "Integrated Services Digital Network (ISDN) User Part (ISUP) to Session Initiation Protocol (SIP) Mapping", RFC 3398, December 2002.

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

[4] Jennings, C. and R. Mahy, "Managing Client Initiated Connections in the Session Initiation Protocol (SIP)", draft-ietf-sip-outbound-04 (work in progress), June 2006.

Author's Address

Jonathan Rosenberg
Cisco Systems
600 Lanidex Plaza
Parsippany, NJ 07054
US

Phone: +1 973 952-5000
Email: jdrosen@cisco.com
URI: http://www.jdrosen.net

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.