Internet Draft
draft-cheng-tcpm-fastopen-00.txt
Intended status: Experimental
Creation date: March 7, 2011
Expiration date: September 8, 2011

Y. Cheng
J. Chu
S. Radhakrishnan
A. Jain
Google, Inc.

TCP Fast Open

Status of this Memo

Distribution of this memo is unlimited.

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on September 8, 2011.

Copyright Notice

Abstract

TCP Fast Open (TFO) allows data to be carried in the SYN or SYN-ACK packets and consumed by the receiving end during the initial connection handshake, thus providing a saving of up to one full round trip time (RTT) compared to standard TCP requiring a three-way handshake (3WHS) to complete before data can be exchanged.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119]. TFO refers to TCP Fast Open. Client refers to the TCP's active open side and server refers to the TCP's passive open side.

1. Introduction

TCP Fast Open (TFO) enables data to be exchanged safely during TCP connection handshake.

This document describes a design that enables qualified applications to attain a round trip saving while avoiding severe security ramifications. At the core of TFO is a security cookie used by the server side to authenticate a client initiating a TFO connection. The document covers the details of exchanging data during TCP's initial handshake, the protocol for TFO cookies, and potential new security vulnerabilities and their mitigation. It also includes discussions on deployment issues and related proposals. TFO requires extensions to the existing socket API, which will be covered in a separate document.

TFO is motivated by the performance need of today's web applications. Network latency is determined by the round-trip time (RTT) and the number of round trips required to transfer application data. RTT consists of transmission delay and propagation delay. Network bandwidth has grown substantially over the past two decades, much reducing the transmission delay, while propagation delay is largely constrained by the speed of light and has remained unchanged. Therefore reducing the number of round trips has become the most effective way to improve the latency of web applications [CDCM10].

Standard TCP only permits data exchange after 3WHS [RFC793], which introduces one RTT delay to the network latency. For short transfers, e.g., web objects, this additional RTT becomes a significant portion of the network latency [THK98]. One widely deployed solution is HTTP persistent connections. However, this solution is limited since hosts and middle boxes terminate idle TCP connections due to resource

constraints. E.g., the Chrome browser keeps TCP connections idle up to 4 minutes but 35% of Chrome HTTP requests are made on new TCP connections.

2. Data In SYN

[RFC793] (section 3.4) already allows data in SYN packets but forbids the receiver to deliver the data to the application until 3WHS is completed. This is because TCP's initial handshake serves to capture

- Old or duplicate SYNs

- SYNs with spoofed IP addresses

TFO allows data to be delivered to the application before 3WHS is completed, thus opening itself to a possible data integrity problem caused by the dubious SYN packets above.

2.1. TCP Semantics and Duplicate SYNs

A past proposal called T/TCP employs a new TCP "TAO" option and connection count to guard against old or duplicate SYNs [RFC1644]. The solution is complex, involving state tracking on per remote peer basis, and is vulnerable to IP spoofing attack. Moreover, it has been shown that even with all the complexity, T/TCP is still not 100% bullet proof. Old or duplicate SYNs may still slip through and get accepted by a T/TCP server [PHRACK98].

Rather than trying to capture all the dubious SYN packets to make TFO 100% compatible with TCP semantics, we've made a design decision early on to accept old SYN packets with data, i.e., to allow TFO for a class of applications that are tolerant of duplicate SYN packets with data, e.g., idempotent or query type transactions. We believe this is the right design trade-off balancing complexity with usefulness. There is a large class of applications that can tolerate dubious transaction requests.

For this reason, TFO MUST be disabled by default, and only enabled explicitly by applications on a per service port basis.

2.2. SYNs with spoofed IP addresses

Standard TCP suffers from the SYN flood attack [RFC4987] because bogus SYN packets, i.e., SYN packets with spoofed source IP addresses can easily fill up a listener's small queue, causing a service port to be blocked completely until timeouts. Secondary damage comes from faked SYN requests taking up memory space. This is normally not an issue today with typical servers having plenty of memory.

TFO goes one step further to allow server side TCP to process and send up data to the application layer before 3WHS is completed. This opens up much more serious new vulnerabilities. Applications serving ports that have TFO enabled may waste lots of CPU and memory resources processing the requests and producing the responses. If the response is much larger than the request, the attacker can mount an amplified reflection attack against victims of choice beyond the TFO server itself.

Numerous mitigation techniques against the regular SYN flood attack exist and have been well documented [RFC4987]. Unfortunately none are applicable to TFO. We propose a server supplied cookie to mitigate most of the security risks introduced by TFO. A more thorough discussion on SYN flood attack against TFO is deferred to the "Security Considerations" section.

3. Protocol Overview

The key component of TFO is the Fast Open Cookie (cookie), a message authentication code (MAC) tag generated by the server. The client requests a cookie in one regular TCP connection, then uses it for future TCP connections to exchange data during 3WHS:

Requesting Fast Open Cookie:

1. The client sends a SYN with a Fast Open Cookie Request option.
2. The server generates a cookie and sends it through the Fast Open Cookie option of a SYN-ACK packet.
3. The client caches the cookie for future TCP Fast Open connections (see below).

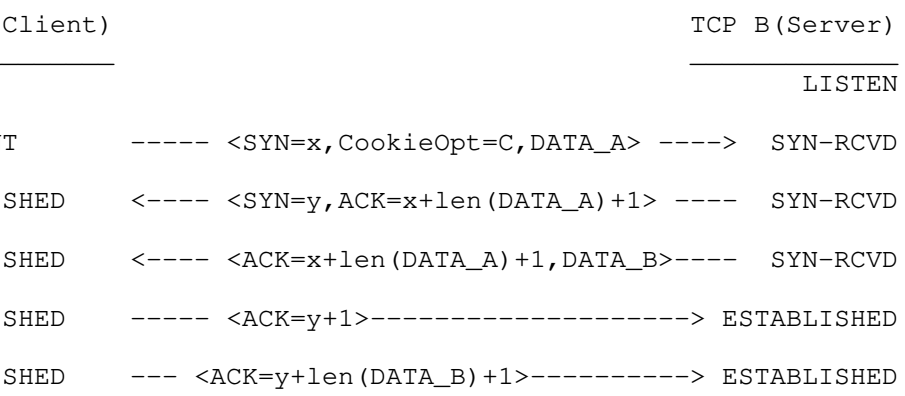
Performing TCP Fast Open:

1. The client sends a SYN with Fast Open Cookie option and data.
2. The server validates the cookie:
   a. If the cookie is valid, the server sends a SYN-ACK acknowledging both the SYN and the data. The server then delivers the data to the application.
   b. Otherwise, the server drops the data and sends a SYN-ACK acknowledging only the SYN sequence number.
3. If the server accepts the data in the SYN packet, it may send the response data before the handshake finishes. The max amount is governed by the TCP's congestion control [RFC5681].
4. The client sends an ACK acknowledging the SYN and the server data. If the client's data is not acknowledged, the client retransmits the data in the ACK packet.
5. The rest of the connection proceeds like a normal TCP connection.

The client can perform many TFO operations once it acquires a cookie until the cookie is expired by the server. Thus TFO is useful for applications that have temporal locality on client and server connections.

Requesting Fast Open Cookie in connection 1:

Performing TCP Fast Open in connection 2:



4. Protocol Details

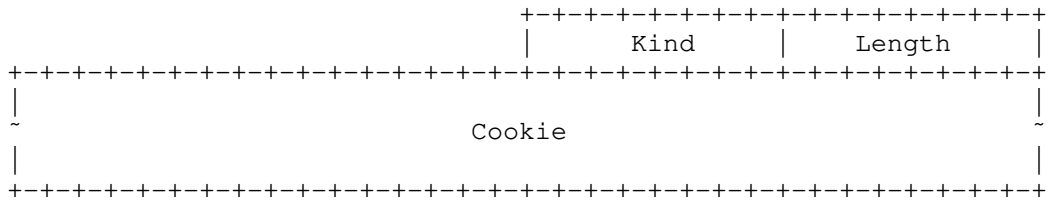
4.1. Fast Open Cookie

The Fast Open Cookie is invented to mitigate new security vulnerabilities in order to enable data exchange during handshake. The cookie is a message authentication code tag generated by the server and is opaque to the client; the client simply caches the cookie and passes it back on subsequent SYN packets to open new connections. The server can expire the cookie at any time to enhance security.

4.1.1. TCP Options

Fast Open Cookie Option

The server uses this option to grant a cookie to the client in the SYN-ACK packet; the client uses it to pass the cookie back to the server in the SYN packet.

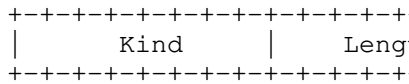


Kind	1 byte: constant TBD (assigned by IANA)
Length	1 byte: range 6 to 18 (bytes); limited by remaining space in the options field. The number MUST be even.
Cookie	4 to 16 bytes (Length - 2)

Options with invalid Length values or without SYN flag set MUST be ignored. The minimum Cookie size is 4 bytes. Although the diagram shows a cookie aligned on 32-bit boundaries, that is not required.

Fast Open Cookie Request Option

The client uses this option in the SYN packet to request a cookie from a TFO-enabled server



Kind	1 byte: same as the Fast Open Cookie option
Length	1 byte: constant 2. This distinguishes the option from the Fast Open cookie option.

Options with invalid Length values, without SYN flag set, or with ACK flag set MUST be ignored.

4.1.2. Server Cookie Handling

The server is in charge of cookie generation and authentication. The cookie SHOULD be a message authentication code tag with the following properties:

1. The cookie authenticates the client's (source) IP address of the SYN packet. The IP address can be an IPv4 or IPv6 address.

2. The cookie can only be generated by the server and can not be fabricated by any other parties including the client.

3. The cookie expires after a certain amount of time. The reason is detailed in the "Security Consideration" section. This can be done by either periodically changing the server key used to generate cookies or including a timestamp in the cookie.

4. The generation and verification are fast relative to the rest of SYN and SYN-ACK processing.

5. A server may encode other information in the cookie, and allow more than one valid cookie per client at any given time. But this is all server implementation dependent and transparent to the client. A client only needs to remember one valid cookie per server IP.

The server supports the cookie generation and verification operations:

- GetCookie(IP_Address): returns a (new) cookie

- IsCookieValid(IP_Address, Cookie): checks if the cookie is valid, i.e., it has not expired and it authenticates the client IP address.

Example Implementation: a simple implementation is to use AES_128 to encrypt the IPv4 (with padding) or IPv6 address and truncate to 64 bits. The server can periodically update the key to expire the cookies. AES encryption on recent processors is fast and takes only a few hundred nanoseconds.

4.1.3. Client Cookie Handling

The client MUST cache cookies from different servers for later Fast Open connections. For a multi-homed client, the cookies are both client and server IP dependent. Beside the cookie, we RECOMMEND that the client caches the MSS and RTT to the server to enhance performance.

The MSS advertised by the server is stored in the cache to determine the maximum amount of data that can be supported in the SYN packet. This information is needed because data is sent before the server announces its MSS in the SYN-ACK packet. Without this information, the data size in the SYN packet is limited to the default MSS of 536 bytes [RFC1122].

Caching RTT allows seeding a more accurate SYN timeout than the default value [RFC2988]. This lowers the performance penalty if the network or the server drops the SYN packets with data or the cookie options (See "Reliability and Deployment Issues" section below).

The cache replacement algorithm is not specified and is left for the implementations.

4.2. Fast Open Protocol

One predominant requirement of TFO is to be fully compatible with existing TCP implementations, both on the client and the server sides.

The server keeps two variables per listening port:

FastOpenEnabled: default is off. It MUST be turned on explicitly by the application. When this flag is off, the server does not perform any TFO related operations and MUST ignore all cookie options.

PendingFastOpenRequests: tracks number of TFO connections in SYN-RCVD state. If this variable goes over the system limit, the server SHOULD set FastOpenEnabled off. This variable is used for defending some vulnerabilities discussed in the "Security Considerations" section.

The server keeps a FastOpened flag per TCB to mark if a connection has successfully performed a TFO.

4.2.1. Fast Open Cookie Request

Any client attempting TFO MUST first request a cookie from the server with the following steps:

1. The client sends a SYN packet with a Fast Open Cookie Request

option.

2. The server SHOULD respond with a SYN-ACK based on the procedures in the "Server Cookie Handling" section. This SYN-ACK SHOULD contain a Fast Open Cookie option if the server currently supports TFO for this listener port.

3. If the SYN-ACK contains a valid Fast Open Cookie option, the client replaces the cookie and other information as described in the "Client Cookie Handling" section. Otherwise, if the SYN-ACK is first seen, i.e.,not a (spurious) retransmission, the client MAY remove the server information from the cookie cache. If the SYN-ACK is a spurious retransmission without valid Fast Open Cookie Option, the client does nothing to the cookie cache for the reasons below.

The network or servers may drop the SYN or SYN-ACK packets with the new cookie options which causes SYN or SYN-ACK timeouts. We RECOMMEND both the client and the server retransmit SYN and SYN-ACK without the cookie options on timeouts. This ensures the connections of cookie requests will go through and lowers the latency penalties (of dropped SYN/SYN-ACK packets). The obvious downside for maximum compatibility is that any regular SYN drop will fail the cookie. We also RECOMMEND the client to record servers that failed to respond to cookie requests and only attempt another cookie request after certain period.

4.2.2. TCP Fast Open

Once the client obtains the cookie from the target server, the client can perform subsequent TFO connections until the cookie is expired by the server. The nature of TCP sequencing makes the TFO specific changes relatively small in addition to [RFC793].

Client: Sending SYN

To open a TFO connection, the client MUST have obtained the cookie from the server:

1. Send a SYN packet.

   a. If the SYN packet does not have enough option space for the Fast Open Cookie option, abort TFO and fall back to regular 3WHS.

   b. Otherwise, include the Fast Open Cookie option with the cookie of the server.Include any data up to the cached server MSS or default 536 bytes.

2. Advance to SYN-SENT state and update SND.NXT to include the data accordingly.

3. If RTT is available from the cache, seed SYN timer according to [RFC2988].

To deal with network or servers dropping SYN packets with payload or unknown options, when the SYN timer fires, the client SHOULD retransmit a SYN packet without data and Fast Open Cookie options.

Server: Receiving SYN and responding with SYN-ACK

Upon receiving the SYN packet with Fast Open Cookie option:

1. If the cookie is invalid, i.e., the cookie does not authenticate the source IP address of the SYN packet, send a SYN-ACK packet acknowledging only the SYN sequence. In addition, include a Fast Open Cookie Option with a new cookie. Go to step 7.

2. If PendingFastOpenRequests is over the system limit, reset FastOpenEnabled flag and send a SYN-ACK acknowledging only the SYN sequence. Go to step 7.

3. Send the SYN-ACK packet acknowledging the SYN and data sequence. The server MAY include data in the SYN-ACK packet.

4. Buffer the data and notify the application.

5. Set FastOpened flag and increment PendingFastOpenRequests.

6. The server MAY send more data packets before the handshake completes. The maximum amount is ruled by the initial congestion window and the receiver window [RFC3390].

7. Advance to the SYN-RCVD state.

If the SYN-ACK timer fires, the server SHOULD retransmit a SYN-ACK packet without data and Fast Open Cookie options for compatibility reasons.

Client: Receiving SYN-ACK

The client SHOULD perform the following steps upon receiving the SYN-ACK:

1. Update the cookie cache if the SYN-ACK has a Fast Open Cookie Option.

2. Send an ACK packet. Set acknowledgment number to RCV.NXT and

include the data after SND.UNA if data is available

3. Advance to the ESTABLISHED state

Note there is no latency penalty if the server does not acknowledge the data in the original SYN packet. The client will retransmit it in the ACK packet. The data exchange will start after the handshake like a regular TCP connection.

Server: Receiving ACK

Upon receiving an ACK acknowledging the SYN sequence, the server decrements PendingFastOpenRequests and advances to the ESTABLISHED state. No special handling is required further.

5. Reliability and Deployment Issues

Network or Hosts Dropping SYN packets with data or unknown options

A study [MAF04] found that some middle-boxes and end-hosts may drop packets with unknown TCP options incorrectly. Another study [LANGLEY06] found that 6% of the probed paths on the Internet drop SYN packets with data. The TFO protocol deals with this problem by retransmitting SYN without data or cookie options and we recommend tracking these servers in the client.

Server Farms

A common server-farm setup is to have many physical hosts behind a load-balancer sharing the same server IP. The load-balancer forwards new TCP connections to different physical hosts based on certain load-balancing algorithms. For TFO to work, the physical hosts need to share the same key and update the key at about the same time.

Network Address Translation (NAT)

The hosts behind NAT sharing same IP address will get the same cookie to the same server. This will not prevent TFO from working. But on some carrier-grade NAT configurations where every new TCP connection from the same physical host uses a different public IP address, TFO does not provide latency benefit. However, there is no performance penalty either as described in Section "Client receiving SYN-ACK".

6. Security Considerations

The Fast Open cookie stops an attacker from trivially flooding spoofed SYN packets with data to burn server resources or to mount an amplified reflection attack on random hosts. The server can defend

against spoofed SYN floods with invalid cookies using existing techniques [RFC4987].

However, the attacker may still obtain cookies from some compromised hosts, then flood spoofed SYN with data and "valid" cookies (from these hosts or other vantage points). With DHCP, it's possible to obtain cookies of past IP addresses without compromising any host. Below we identify two new vulnerabilities of TFO and describe the countermeasures.

6.1. Server Resource Exhaustion Attack by SYN Flood with Valid Cookies

Like regular TCP handshakes, TFO is vulnerable to such an attack. But the potential damage can be much more severe. Besides causing temporary disruption to service ports under attack, it may exhaust server CPU and memory resources.

For this reason it is crucial for the TFO server to limit the maximum number of total pending TFO connection requests, i.e., PendingFastOpenRequests. When the limit is exceeded, the server temporarily disables TFO entirely as described in "Server Cookie Handling". Then subsequent TFO requests will be downgraded to regular connection requests, i.e., with the data dropped and only SYN acknowledged. This allows regular SYN flood defense techniques [RFC4987] like SYN-cookies to kick in and prevent further service disruption.

There are other subtle but important differences in the vulnerability between TFO and regular TCP handshake. Before the SYN flood attack broke out in the late '90s, typical listener's max qlen was small, enough to sustain the highest expected new connection rate and the average RTT for the SYN-ACK packets to be acknowledged in time. E.g., if a server is designed to handle at most 100 connection requests per second, and the average RTT is 100ms, a max qlen on the order of 10 will be sufficient.

This small max qlen made it very easy for any attacker, even equipped with just a dailup modem to the Internet, to cause major disruptions to a web site by simply throwing a handful of "SYN bombs" at its victim of choice. But for this attack scheme to work, the attacker must pick a non-responsive source IP address to spoof with. Otherwise the SYN-ACK packet will trigger TCP RST from the host whose IP address has been spoofed, causing corresponding connection to be removed from the server's listener queue hence defeating the attack. In other words, the main damage of SYN bombs against the standard TCP stack is not directly from the bombs themselves costing TCP processing overhead or host memory, but rather from the spoofed SYN packets filling up the often small listener's queue.

On the other hand, TFO SYN bombs can cause damage directly if admitted without limit into the stack. The RST packets from the spoofed host will fuel rather than defeat the SYN bombs as compared to the non-TFO case, because the attacker can flood more SYNs with data to cost more data processing resources. For this reason, a TFO server needs to monitor the connections in SYN-RCVD being reset in addition to imposing a reasonable max qlen. Implementations may combine the two, e.g., by continuing to account for those connection requests that have just been reset against the listener's PendingFastOpenRequests until a timeout period has passed.

Limiting the maximum number of pending TFO connection requests does make it easy for an attacker to overflow the queue, causing TFO to be disabled. We argue that causing TFO to be disabled is unlikely to be of interest to attackers because the service will remain intact without TFO hence there is hardly any real damage.

6.2. Amplified Reflection Attack to Random Host

Limiting PendingFastOpenRequests with a system limit can be done without Fast Open Cookies and would protect the server from resource exhaustion. It would also limit how much damage an attacker can cause through an amplified reflection attack from that server. However, it would still be vulnerable to an amplified reflection attack from a large number of servers. An attacker can easily cause damage by tricking many servers to respond with data packets at once to any spoofed victim IP address of choice.

With the use of Fast Open Cookies, the attacker would first have to steal a valid cookie from its target victim. This likely requires the attacker to compromise the victim host or network first.

The attacker here has little interest in mounting an attack on the victim host that has already been compromised. But she may be motivated to disrupt the victim's network. Since a stolen cookie is only valid for a single server, she has to steal valid cookies from a large number of servers and use them before they expire to cause sufficient damage without triggering the defense in the previous section.

One can argue that if the attacker has compromised the target network or hosts, she could perform a similar but simpler attack by injecting bits directly. The degree of damage will be identical, but TFO-specific attack allows the attacker to remain anonymous and disguises the attack as from other servers.

The best defense is for the server to not respond with data until handshake finishes, i.e., disallow step 6 in "Server receiving SYN-

ACK" section. In this case the risk of amplification reflection attack is completely eliminated, but the potential latency saving from TFO may diminish if the server application produces responses earlier before the handshake completes.

7. Related Work

7.1. T/TCP

TCP Extensions for Transactions [RFC1644] attempted to bypass the three-way handshake, among other things, hence shared the same goal but also the same set of issues as TFO. It focused most of its effort battling old or duplicate SYNs, but paid no attention to security vulnerabilities it introduced when bypassing 3WHS. Its TAO option and connection count, besides adding complexity, require the server to keep state per remote host, while still leaving it wide open for attacks. It is trivial for an attacker to fake a CC value that will pass the TAO test. Unfortunately, in the end its scheme is still not 100% bullet proof as pointed out by [PHRACK98].

As stated earlier, we take a practical approach to focus TFO on the security aspect, while allowing old, duplicate SYN packets with data after recognizing that 100% TCP semantics is likely infeasible. We believe this approach strikes the right tradeoff, and makes TFO much simpler and more appealing to TCP implementers and users.

7.2. Common Defenses Against SYN Flood Attacks

TFO is still vulnerable to SYN flood attacks just like normal TCP handshakes, but the damage may be much worse, thus deserves a careful thought.

There have been plenty of studies on how to mitigate attacks from regular SYN flood, i.e., SYN without data [RFC4987]. But from the stateless SYN-cookies to the stateful SYN Cache, none can preserve data sent with SYN safely while still providing an effective defense.

The best defense may be to simply disable TFO when a host is suspected to be under a SYN flood attack, e.g., the SYN backlog is filled. Once TFO is disabled, normal SYN flood defenses can be applied. The "Security Consideration" section contains a thorough discussion on this topic.

7.3. TCP Cookie Transaction (TCPCT)

TCPCT [RFC6013] eliminates server state during initial handshake and defends spoofing DoS attacks. Like TFO, TCPCT allows SYN and SYN-ACK packets to carry data. However, TCPCT and TFO are designed for

different goals and they are not compatible.

The TCPCT server does not keep any connection state during the handshake, therefore the server application needs to consume the data in SYN and (immediately) produce the data in SYN-ACK before sending SYN-ACK. Otherwise the application's response has to wait until handshake completes. In contrary, TFO allows server to respond data during handshake. Therefore for many request-response style applications, TCPCT may not achieve same latency benefit as TFO.

Without state kept on the server side, TCPCP relies on the client side to retransmit the SYN request with data in order to recover from possible loss of packet from server response. This may cause a lot more dubious connection requests. It also limits the response to only one packet, to fit completely within the SYN-ACK packet. For some TCP applications, in particular web applications, this does not provide enough latency benefit by sending one data packet one RTT earlier.

8. IANA Considerations

The Fast Open Cookie Option and Fast Open Cookie Request Option define no new namespace. The options require IANA allocate one value from the TCP option Kind namespace.

9. Acknowledgements

The authors would like to thank Tom Herbert, Adam Langley, Roberto Peon, Mathew Mathis, and Barath Raghavan for their insightful comments.

10. References

10.1. Normative References

[RFC793] Postel, J. "Transmission Control Protocol", RFC 793, September 1981.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[RFC5681] Allman, M., Paxson, V. and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

10.2. Informative References

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, August 2007.

[RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC6013, January 2011.

[CDCM10] Chu, J., Dukkipati, N., Cheng, Y. and M. Mathis, "Increasing TCP's Initial Window", Internet-Draft draft-ietf-tcpm-initcwnd-00.txt (work in progress), October 2010.

[THK98] Touch, J., Heidemann, J., Obraczka, K., "Analysis of HTTP Performance", USC/ISI Research Report 98-463. December 1998.

[PHRACK98] "T/TCP vulnerabilities", Phrack Magazine, Volume 8, Issue 53 artical 6. July 8, 1998. URL http://www.phrack.com/issues.html?issue=53&id=6

[MAF04] Medina, A., Allman, M., and S. Floyd, "Measuring Interactions Between Transport Protocols and Middleboxes", Proceedings 4th ACM SIGCOMM/USENIX Conference on Internet Measurement, October 2004.

[LANGLEY06] Langley, A, "Probing the viability of TCP extensions", URL http://www.imperialviolet.org/binary/ecntest.pdf

Author's Addresses

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: ycheng@google.com

H.K. Jerry Chu
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: hkchu@google.com

Sivasankar Radhakrishnan
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: sivasankar@google.com

Arvind Jain
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: arvind@google.com

Acknowledgement

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

Internet Draft
draft-ietf-tcpm-initcwnd-01.txt
Intended status: Standard
Updates: 3390, 5681
Creation date: April 15, 2011
Expiration date: October 2011

J. Chu
N. Dukkipati
Y. Cheng
M. Mathis
Google, Inc.

Increasing TCP's Initial Window

Status of this Memo

Copyright Notice

Abstract

This document proposes an increase in the permitted TCP initial window (IW) from between 2 and 4 segments, as specified in RFC 3390, to 10 segments. It discusses the motivation behind the increase, the advantages and disadvantages of the higher initial window, and presents results from several large scale experiments showing that the higher initial window improves the overall performance of many web services without risking congestion collapse. The document closes with a discussion of a list of concerns, and some results from recent studies to address the concerns.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Table of Contents

1. Introduction 2
2. TCP Modification 3
3. Implementation Issues 4
4. Background 5
5. Advantages of Larger Initial Windows 6
5.1 Reducing Latency 6
5.2 Keeping up with the growth of web object size 7
5.3 Recovering faster from loss on under-utilized or wireless links 7
7. Disadvantages of Larger Initial Windows for the Network 9
8. Mitigation of Negative Impact 9
9. Interactions with the Retransmission Timer 9
10. Experimental Results From Large Scale Cluster Tests 10
10.1 The benefits 10
10.2 The cost 11
11. List of Concerns and Corresponding Test Results 12
12. Related Proposals 14
14. Conclusion 15
15. IANA Considerations 15
16. Acknowledgments 15
Normative References 16
Informative References 16
Author's Addresses 20
Acknowledgement 20

1. Introduction

This document updates RFC 3390 to raise the upper bound on TCP's initial window (IW) to 10 segments or roughly 15KB. It is patterned after and borrows heavily from RFC 3390 [RFC3390] and earlier work in this area.

The primary argument in favor of raising IW follows from the evolving scale of the Internet. Ten segments are likely to fit into queue space available at any broadband access link, even when there are a reasonable number of concurrent connections.

Lower speed links can be treated with environment specific configurations, such that they can be protected from being overwhelmed by large initial window bursts without imposing a suboptimal initial window on the rest of the Internet.

This document reviews the advantages and disadvantages of using a larger initial window, and includes summaries of several large scale experiments showing that an initial window of 10 segments provides benefits across the board for a variety of BW, RTT, and BDP classes. These results show significant benefits for increasing IW for users at much smaller data rates than had been previously anticipated. However, at initial windows larger than 10, the results are mixed. We believe that these mixed results are not intrinsic, but are the consequence of various implementation artifacts, including overly aggressive applications employing many simultaneous connections.

We propose that all TCP implementations should have a settable TCP IW parameter; the default setting may start at 10 segments and should be raised as we come to understand and and correct things that conflict.

In addition, we introduce a minor revision to RFC 3390 and RFC 5681 [RFC5681] to eliminate resetting the initial window when the SYN or SYN/ACK is lost.

The document closes with a discussion of a list of concerns that have been brought up, and some recent test results showing most of the concerns can not be validated.

A complementary set of slides for this proposal can be found at [CD10].

2. TCP Modification

This document proposes an increase in the permitted upper bound for TCP's initial window (IW) to 10 segments. This increase is optional: a TCP MAY start with a larger initial window up to 10 segments.

This upper bound for the initial window size represents a change from

RFC 3390 [RFC3390], which specified that the congestion window be initialized between 2 and 4 segments depending on the MSS.

This change applies to the initial window of the connection in the first round trip time (RTT) of data transmission following the TCP three-way handshake. Neither the SYN/ACK nor its acknowledgment (ACK) in the three-way handshake should increase the initial window size.

Furthermore, RFC 3390 and RFC 5681 [RFC5681] state that

> "If the SYN or SYN/ACK is lost, the initial window used by a sender after a correctly transmitted SYN MUST be one segment consisting of MSS bytes."

The proposed change to reduce the default RTO to 1 second [PACS11] increases the chance for spurious SYN or SYN/ACK retransmission, thus unnecessarily penalizing connections with RTT > 1 second if their initial window is reduced to 1 segment. For this reason, it is RECOMMENDED that implementations refrain from resetting the initial window to 1 segment, unless either there have been multiple SYN or SYN/ACK retransmissions, or true loss detection has been made.

TCP implementations use slow start in as many as three different ways: (1) to start a new connection (the initial window); (2) to restart transmission after a long idle period (the restart window); and (3) to restart transmission after a retransmit timeout (the loss window). The change specified in this document affects the value of the initial window. Optionally, a TCP MAY set the restart window to the minimum of the value used for the initial window and the current value of cwnd (in other words, using a larger value for the restart window should never increase the size of cwnd). These changes do NOT change the loss window, which must remain 1 segment of MSS bytes (to permit the lowest possible window size in the case of severe congestion).

Furthermore, to limit any negative effect that a larger initial window may have on links with limited bandwidth or buffer space, implementations SHOULD fall back to RFC 3390 for the restart window (RW), if any packet loss is detected during either the initial window, or a restart window, when more than 4KB of data is sent.

3. Implementation Issues

[Need to decide if a different formula is needed for PMTU != 1500.]

HTTP 1.1 specification allows only two simultaneous connections per domain, while web browsers open more simultaneous TCP connections [Ste08], partly to circumvent the small initial window in order to

speed up the loading of web pages as described above.

When web browsers open simultaneous TCP connections to the same destination, they are working against TCP's congestion control mechanisms [FF99]. Combining this behavior with larger initial windows further increases the burstiness and unfairness to other traffic in the network. A larger initial window will incentivize applications to use fewer concurrent TCP connections.

Some implementations advertise small initial receive window (Table 2 in [Duk10]), effectively limiting how much window a remote host may use. In order to realize the full benefit of the large initial window, implementations are encouraged to advertise an initial receive window of at least 10 segments, except for the circumstances where a larger initial window is deemed harmful. (See the Mitigation section below.)

TCP SACK option ([RFC2018]) was thought to be required in order for the larger initial window to perform well. But measurements from both a testbed and live tests showed that IW=10 without the SACK option still beats the performance of IW=3 with the SACK option [CW10].

4. Background

TCP congestion window was introduced as part of the congestion control algorithm by Van Jacobson in 1988 [Jac88]. The initial value of one segment was used as the starting point for newly established connections to probe the available bandwidth on the network.

Today's Internet is dominated by web traffic running on top of short-lived TCP connections [IOR2009]. The relatively small initial window has become a limiting factor for the performance of many web applications.

The global Internet has continued to grow, both in speed and penetration. According to the latest report from Akamai [AKAM10], the global broadband (> 2Mbps) adoption has surpassed 50%, propelling the average connection speed to reach 1.7Mbps, while the narrowband (< 256Kbps) usage has dropped to 5%. In contrast, TCP's initial window has remained 4KB for a decade [RFC2414], corresponding to a bandwidth utilization of less than 200Kbps per connection, assuming an RTT of 200ms.

A large proportion of flows on the Internet are short web transactions over TCP, and complete before exiting TCP slow start. Speeding up the TCP flow startup phase, including circumventing the initial window limit, has been an area of active research [PWSB09, Sch08]. Numerous proposals exist [LAJW07, RFC4782, PRAKS02, PK98].

Some require router support [RFC4782, PK98], hence are not practical for the public Internet. Others suggested bold, but often radical ideas, likely requiring more years of research before standardization and deployment.

In the mean time, applications have responded to TCP's "slow" start. Web sites use multiple sub-domains [Bel10] to circumvent HTTP 1.1 regulation on two connections per physical host [RFC2616]. As of today, major web browsers open multiple connections to the same site (up to six connections per domain [Ste08] and the number is growing). This trend is to remedy HTTP serialized download to achieve parallelism and higher performance. But it also implies today most access links are severely under-utilized, hence having multiple TCP connections improves performance most of the time. While raising the initial congestion window may cause congestion for certain users using these browsers, we argue that the browsers and other application need to respect HTTP 1.1 regulation and stop increasing number of simultaneous TCP connections. We believe a modest increase of the initial window will help to stop this trend, and provide the best interim solution to improve overall user performance, and reduce the server, client, and network load.

Note that persistent connections and pipelining are designed to address some of the issues with HTTP above [RFC2616]. Their presence does not diminish the need for a larger initial window. E.g., data from the Chrome browser show that 35% of HTTP requests are made on new TCP connections. Our test data also confirm significant latency reduction with the large initial window even with these two HTTP features ([Duk10]).

Also note that packet pacing has been suggested as an effective mechanism to avoid large bursts and their associated damage [VH97]. We do not require pacing in our proposal due to our strong preference for a simple solution. We suspect for packet bursts of a moderate size, packet pacing will not be necessary. This seems to be confirmed by our test results.

More discussion of the increase in initial window, including the choice of 10 segments can be found in [Duk10, CD10].

5. Advantages of Larger Initial Windows

5.1 Reducing Latency

An increase of the initial window from 3 segments to 10 segments reduces the total transfer time for data sets greater than 4KB by up to 4 round trips.

The table below compares the number of round trips between IW=3 and IW=10 for different transfer sizes, assuming infinite bandwidth, no packet loss, and the standard delayed acks with large delayed-ack timer.

total segments	IW=3	IW=10
3	1	1
6	2	1
10	3	1
12	3	2
21	4	2
25	5	2
33	5	3
46	6	3
51	6	4
78	7	4
79	8	4
120	8	5
127	9	5

For example, with the larger initial window, a transfer of 32 segments of data will require only two rather than five round trips to complete.

5.2 Keeping up with the growth of web object size

RFC 3390 stated that the main motivation for increasing the initial window to 4KB was to speed up connections that only transmit a small amount of data, e.g., email and web. The majority of transfers back then were less than 4KB, and could be completed in a single RTT [All00].

Since RFC 3390 was published, web objects have gotten significantly larger [Chu09, RJ10]. Today only a small percentage of web objects (e.g., 10% of Google's search responses) can fit in the 4KB initial window. The average HTTP response size of gmail.com, a highly scripted web-site, is 8KB (Figure 1. in [Duk10]). The average web page, including all static and dynamic scripted web objects on the page, has seen even greater growth in size [RJ10]. HTTP pipelining [RFC2616] and new web transport protocols like SPDY [SPDY] allow multiple web objects to be sent in a single transaction, potentially requiring even larger initial window in order to transfer a whole web page in one round trip.

5.3 Recovering faster from loss on under-utilized or wireless links

A greater-than-3-segment initial window increases the chance to recover packet loss through Fast Retransmit rather than the lengthy initial RTO [RFC5681]. This is because the fast retransmit algorithm requires three duplicate acks as an indication that a segment has been lost rather than reordered. While newer loss recovery techniques such as Limited Transmit [RFC3042] and Early Retransmit [RFC5827] have been proposed to help speeding up loss recovery from a smaller window, both algorithms can still benefit from the larger initial window because of a better chance to receive more ACKs to react upon.

6. Disadvantages of Larger Initial Windows for the Individual Connection

The larger bursts from an increase in the initial window may cause buffer overrun and packet drop in routers with small buffers, or routers experiencing congestion. This could result in unnecessary retransmit timeouts. For a large-window connection that is able to recover without a retransmit timeout, this could result in an unnecessarily-early transition from the slow-start to the congestion-avoidance phase of the window increase algorithm. [Note: knowing the large initial window may cause premature segment drop, should one make an exception for it, i.e., by allowing ssthresh to remain unchanged if loss is from an enlarged initial window?]

Premature segment drops are unlikely to occur in uncongested networks with sufficient buffering, or in moderately-congested networks where the congested router uses active queue management (such as Random Early Detection [FJ93, RFC2309, RFC3150]).

Insufficient buffering is more likely to exist in the access routers connecting slower links. A recent study of access router buffer size [DGHS07] reveals the majority of access routers provision enough buffer for 130ms or longer, sufficient to cover a burst of more than 10 packets at 1Mbps speed, but possibly not sufficient for browsers opening simultaneous connections.

A testbed study [CW10] on the effect of the larger initial window with five simultaneously opened connections revealed that, even with limited buffer size on slow links, IW=10 still reduced the total latency of web transactions, although at the cost of higher packet drop rates as compared to IW=3.

Some TCP connections will receive better performance with the larger initial window even if the burstiness of the initial window results in premature segment drops. This will be true if (1) the TCP connection recovers from the segment drop without a retransmit timeout, and (2) the TCP connection is ultimately limited to a small congestion window by either network congestion or by the receiver's advertised window.

7. Disadvantages of Larger Initial Windows for the Network

An increase in the initial window may increase congestion in a network. However, since the increase is one-time only (at the beginning of a connection), and the rest of TCP's congestion backoff mechanism remains in place, it's highly unlikely the increase will render a network in a persistent state of congestion, or even congestion collapse. This seems to have been confirmed by our large scale experiments described later.

Some of the discussions from RFC 3390 are still valid for IW=10. Moreover, it is worth noting that although TCP NewReno increases the chance of duplicate segments when trying to recover multiple packet losses from a large window [RFC3782], the wide support of TCP Selective Acknowledgment (SACK) option [RFC2018] in all major OSes today should keep the volume of duplicate segments in check.

Recent measurements [Get11] provide evidence of extremely large queues (in the order of one second) at access networks of the Internet. While a significant part of the buffer bloat is contributed by large downloads/uploads such as video files, emails with large attachments, backups and download of movies to disk, some of the problem is also caused by Web browsing of image heavy sites [Get11]. This queuing delay is generally considered harmful for responsiveness of latency sensitive traffic such as DNS queries, ARP, DHCP, VoIP and Gaming. IW=10 can exacerbate this problem when doing short downloads such as Web browsing. The mitigations proposed for the broader problem of buffer bloating are also applicable in this case, such as the use of ECN, AQM schemes and traffic classification (QoS).

8. Mitigation of Negative Impact

Much of the negative impact from an increase in the initial window is likely to be felt by users behind slow links with limited buffers. The negative impact can be mitigated by hosts directly connected to a low-speed link advertising a smaller initial receive window than 10 segments. This can be achieved either through manual configuration by the users, or through the host stack auto-detecting the low bandwidth links.

More suggestions to improve the end-to-end performance of slow links can be found in RFC 3150 [RFC3150].

[Note: if packet loss is detected during IW through fast retransmit, should cwnd back down to 2 rather than FlightSize / 2?]

9. Interactions with the Retransmission Timer

A large initial window increases the chance of spurious RTO on a low-bandwidth path because the packet transmission time will dominate the round-trip time. To minimize spurious retransmissions, implementations MUST follow RFC 2988 [RFC2988] to restart the retransmission timer with the current value of RTO for each ack received that acknowledges new data.

10. Experimental Results From Large Scale Cluster Tests

In this section we summarize our findings from large scale Internet experiments with an initial window of 10 segments, conducted via Google's front-end infrastructure serving a diverse set of applications. We present results from two data centers, each chosen because of the specific characteristics of subnets served: AvgDC has connection bandwidths closer to the worldwide average reported in [AKAM10], with a median connection speed of about 1.7Mbps; SlowDC has a larger proportion of traffic from slow bandwidth subnets with nearly 20% of traffic from connections below 100Kbps, and a third below 256Kbps.

Guided by measurements data, we answer two key questions: what is the latency benefit when TCP connections start with a higher initial window, and on the flip side, what is the cost?

10.1 The benefits

The average web search latency improvement over all responses in AvgDC is 11.7% (68 ms) and 8.7% (72 ms) in SlowDC. We further analyzed the data based on traffic characteristics and subnet properties such as bandwidth (BW), round-trip time (RTT), and bandwidth-delay product (BDP). The average response latency improved across the board for a variety of subnets with the largest benefits of over 20% from high RTT and high BDP networks, wherein most responses can fit within the pipe. Correspondingly, responses from low RTT paths experienced the smallest improvements of about 5%.

Contrary to what we expected, responses from low bandwidth subnets experienced the best latency improvements (between 10-20%) in the buckets 0-56Kbps and 56-256Kbps buckets. We speculate low BW networks observe improved latency for two plausible reasons: 1) fewer slow-start rounds: unlike many large BW networks, low BW subnets with dial-up modems have inherently large RTTs; and 2) faster loss recovery: an initial window larger than 3 segments increases the chances of a lost packet to be recovered through Fast Retransmit as opposed to a lengthy RTO.

Responses of different sizes benefited to varying degrees; those larger than 3 segments naturally demonstrated larger improvements,

because they finished in fewer rounds in slow start as compared to the baseline. In our experiments, response sizes <= 3 segments also demonstrated small latency benefits.

To find out how individual subnets performed, we analyzed average latency at a /24 subnet level (an approximation to a user base offered similar set of services by a common ISP). We find even at the subnet granularity, latency improved at all quantiles ranging from 5-11%.

10.2 The cost

To quantify the cost of raising the initial window, we analyzed the data specifically for subnets with low bandwidth and BDP, retransmission rates for different kinds of applications, as well as latency for applications operating with multiple concurrent TCP connections. From our measurements we found no evidence of a negative latency impacts that correlate to BW or BDP alone, but in fact both kinds of subnets demonstrated latency improvements across averages and quantiles.

As expected, the retransmission rate increased modestly when operating with larger initial congestion window. The overall increase in AvgDC is 0.3% (from 1.98% to 2.29%) and in SlowDC is 0.7% (from 3.54% to 4.21%). In our investigation, with the exception of one application, the larger window resulted in a retransmission increase of < 0.5% for services in the AvgDC. The exception is the Maps application that operates with multiple concurrent TCP connections, which increased its retransmission rate by 0.9% in AvgDC and 1.85% in SlowDC (from 3.94% to 5.79%).

In our experiments, the percentage of traffic experiencing retransmissions did not increase significantly. E.g. 90% of web search and maps experienced zero retransmission in SlowDC (percentages are higher for AvgDC); a break up of retransmissions by percentiles indicate that most increases come from portion of traffic already experiencing retransmissions in the baseline with initial window of 3 segments.

Traffic patterns from applications using multiple concurrent TCP connections all operating with a large initial window represent one of the worst case scenarios where latency can be adversely impacted due to bottleneck buffer overflow. Our investigation shows that such a traffic pattern has not been a problem in AvgDC, where all these applications, specifically maps and image thumbnails, demonstrated improved latencies varying from 2-20%. In the case of SlowDC, while these applications continued showing a latency improvement in the mean, their latencies in higher quantiles (96 and above for maps)

indicated instances where latency with larger window is worse than the baseline, e.g. the 99% latency for maps has increased by 2.3% (80ms) when compared to the baseline. There is no evidence from our measurements that such a cost on latency is a result of subnet bandwidth alone. Although we have no way of knowing from our data, we conjecture that the amount of buffering at bottleneck links plays a key role in performance of these applications.

Further details on our experiments and analysis can be found in [Duk10, DCCM10].

11. List of Concerns and Corresponding Test Results

Concerns have been raised since we first published our proposal based on a set of large scale experiments. To better understand the impact of a larger initial window in order to confirm or dismiss these concerns, we, as well as people outside of Google have conducted numerous additional tests in the past year, using either Google's large scale clusters, simulations, or real testbeds. The following is a list of concerns and some of the findings.

A complete list of tests conducted, their results and related studies can be found at [IW10].

- How complete are our tests in traffic pattern coverage?

  Google today offers a large portfolio of services beyond web search. The list includes Gmail, Google Maps, Photos, News, Sites, Images, Videos,..., etc. Our tests included most of Google's services, covering a wide variety of traffic sizes and patterns. One notable exception is YouTube because we don't think the large initial window will have much material impact, either positive or negative, on bulk data services.

  [CW10] contains some result from a testbed study on how short flows with a larger initial window might affect the throughput performance of other co-existing, long lived, bulk data transfers.

- Larger bursts from the increase in the initial window cause significantly more packet drops

  All the known tests conducted on this subject so far [Duk10, Sch11, Sch11-1, CW10] show that, although bursts from the larger initial window tend to cause more packet drops, the increase tends to be very modest. The only exception is from our own testbed study [CW10] when under extremely high load and/or simultaneous opens. But both IW=3 and IW=10 suffered very high packet loss rates under those conditions.

- o A large initial window may severely impact TCP performance over highly multiplexed links still common in developing regions

  Our large scale experiments described in section 10 above also covered Africa and South America. Measurement data from those regions [DCCM10] revealed improved latency even for those Google services that employ multiple simultaneous connections, at the cost of small increase in the retransmission rate. It seems that the round trip savings from a larger initial window more than make up the time spent on recovering more lost packets.

  Similar phenomenon have also been observed from our testbed study [CW10].

- o Why 10 segments?

  Questions have been raised on how the number 10 was picked. We have tried different sizes in our large scale experiments, and found that 10 segments seem to give most of the benefits for the services we tested while not causing significant increase in the retransmission rates. Going forward 10 segments may turn out to be too small when the average of web object sizes continue to grow. A scheme to attempt to right size the initial window automatically over long timescales has been proposed in [Tou10].

- o Need more thorough analysis of the impact on slow links

  Although data from [Duk10] showed the large initial window reduced the average latency even for the dialup link class of only 56Kbps in bandwidth, it is only prudent to perform more microscopic analysis on its effect on slow links. We set up two testbeds for this purpose [CW10].

  Both testbeds were used to emulate a 300ms RTT, bottleneck link bandwidth as low as 64Kbps, and route queue size as low as 40 packets. Although we've tried a large combination of test parameters, almost all tests we ran managed to show some latency improvement from IW=10, with only a modest increase in the packet drop rate until a very high load was injected. The testbed result was consistent with both our own large scale data center experiments [CD10, DCCM10] and a separate study using NSC simulations [Sch11, Sch11-1].

- o How will the larger initial window affect flows with initial windows 4KB or less?

  Flows with the larger initial window will likely grab more bandwidth from a bottleneck link when competing against flows with

smaller initial window, at least initially. How long will this "unfairness" last? Will there be any "capture effect" where flows with larger initial window possess a disproportional share of bandwidth beyond just a few round trips?

If there is any "unfairness" issue from flows with different initial windows, it did not show up in our large scale experiments, as the average latency for the bucket of all responses < 4KB did not seem to be affected by the presence of many other larger responses employing large initial window. As a matter of fact they seemed to benefit from the large initial window too, as shown in Figure 7 of [Duk10].

The same phenomenon seems to exist in our testbed experiments. Flows with IW=3 only suffered slightly when competing against flows with IW=10 in light to median loads. Under high load both flows' latency improved when mixed together. Also long-lived, background bulk-data flows seemed to enjoy higher throughput when running against many foreground short flows of IW=10 than against short flows of IW=3. One plausible explanation was IW=10 enabled short flows to complete sooner, leaving more room for the long-lived, background flows.

An independent study using NSC simulator has also concluded that IW=10 works rather well and is quite fair against IW=3 [Sch11, Sch11-1].

- How will a larger initial window perform over cellular networks?

  Some simulation studies [JNDK10, JNDK10-1] have been conducted to study the effect of a larger initial window on wireless links from 2G to 4G networks (EGDE/HSPA/LTE). The overall result seems mixed in both raw performance and the fairness index.

  There has been on-going studies by people from Nokia on the effect of a larger initial window on GPRS and HSDPA networks. Initial test results seem to show no or little improvement from flows with a larger initial window. More studies are needed to understand why.

12. Related Proposals

Two other proposals [All10, Tou10] have been made with the goal to raise TCP's initial window size over a large timescale. Both aim at addressing the concern about the uncertain impact from raising the initial window size at an Internet wide scale. Moreover, [Tou10] seeks an algorithm to automate the adjustment of IW safely over long haul period.

Based on our test results from the past couple of years, we believe our proposal – a modest, static increase of IW to 10, to be the best near-term solution that is both simple and effective. The other proposals, with their added complexity and much longer deployment cycles, seem best suited for growing IW beyond 10 in the long run.

13. Security Considerations

This document discusses the initial congestion window permitted for TCP connections. Changing this value does not raise any known new security issues with TCP.

14. Conclusion

This document suggests a simple change to TCP that will reduce the application latency over short-lived TCP connections or links with long RTTs (saving several RTTs during the initial slow-start phase) with little or no negative impact over other flows. Extensive tests have been conducted through both testbeds and large data centers with most results showing improved latency with only a small increase in the packet retransmission rate. Based on these results we believe a modest increase of IW to 10 is the best near-term proposal while other proposals [All10, Tou10] may be best suited to grow IW beyond 10 in the long run.

15. IANA Considerations

None

16. Acknowledgments

Many people at Google have helped to make the set of large scale tests possible. We would especially like to acknowledge Amit Agarwal, Tom Herbert, Arvind Jain and Tiziana Refice for their major contributions.

Normative References

[PACS11] Paxson, V., Allman, M., Chu, J. and M. Sargent, "Computing TCP's Retransmission Timer", Internet-draft draft-paxson-tcpm-rfc2988bis-02, work in progress.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S. and A. Romanow, "TCP Selective Acknowledgement Options", RFC 2018, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[RFC3390] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's Initial Window", RFC 3390, October 2002.

[RFC5681] Allman, M., Paxson, V. and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J. and P. Hurtig, "Early Retransmit for TCP and SCTP", RFC 5827, April 2010.

Informative References

[AKAM10] "The State of the Internet, 3rd Quarter 2009", Akamai Technologies, Inc., January 2010.

[All00] Allman, M., "A Web Server's View of the Transport Layer", ACM Computer Communication Review, 30(5), October 2000.

[All10] Allman, M., "Initial Congestion Window Specification", Internet-draft draft-allman-tcpm-bump-initcwnd-00.txt work in progress.

[Bel10] Belshe, M., "A Client-Side Argument For Changing TCP Slow Start", January, 2010. URL http://sites.google.com/a/chromium.org/dev/spdy/An_Argument_For_Changing_TCP_Slow_Start.pdf

[CD10] Chu, J. and N. Dukkipati, "Increasing TCP's Initial Window", Presented to 77th IRTF ICCRG & IETF TCPM working

group meetings, March 2010. URL
http://www.ietf.org/proceedings/77/slides/tcpm-4.pdf

[Chu09] Chu, J., "Tuning TCP Parameters for the 21st Century", Presented to 75th IETF TCPM working group meeting, July 2009. URL http://www.ietf.org/proceedings/75/slides/tcpm-1.pdf.

[CW10] Chu, J. and Wang, Y., "A Testbed Study on IW10 vs IW3", Presented to 79th IETF TCPM working group meeting, Nov. 2010. URL http://www.ietf.org/proceedings/79/slides/tcpm-0.pdf.

[DCCM10] Dukkipati, D., Cheng, Y., Chu, J. and M. Mathis, "Increasing TCP initial window", Presented to 78th IRTF ICCRG working group meeting, July 2010. URL http://www.ietf.org/proceedings/78/slides/iccrg-3.pdf

[DGHS07] Dischinger, M., Gummadi, K., Haeberlen, A. and S. Saroiu, "Characterizing Residential Broadband Networks", Internet Measurement Conference, October 24-26, 2007.

[Duk10] Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Sutin, N., Agarwal, A., Herbert, T. and J. Arvind, "An Argument for Increasing TCP's Initial Congestion Window", ACM SIGCOMM Computer Communications Review, vol. 40 (2010), pp. 27-33. July 2010. URL
http://www.google.com/research/pubs/pub36640.html

[FF99] Floyd, S., and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet", IEEE/ACM Transactions on Networking, August 1999.

[FJ93] Floyd, S. and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413.

[Get11] Gettys, J., "Bufferbloat: Dark buffers in the Internet", Presented to 80th IETF TSV Area meeting, March 2011. URL
http://www.ietf.org/proceedings/80/slides/tsvarea-1.pdf

[IOR2009] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J. Jahanian, F. and M. Karir, "Atlas Internet Observatory 2009 Annual Report", 47th NANOG Conference, October 2009.

[IW10] "TCP IW10 links", URL
http://code.google.com/speed/protocols/tcpm-IW10.html

[Jac88] Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.

[JNDK10] Jarvinen, I., Nyrhinen. A., Ding, A. and M. Kojo, "A Simulation Study on Increasing TCP's IW", Presented to 78th IRTF ICCRG working group meeting, July 2010. URL http://www.ietf.org/proceedings/78/slides/iccrg-7.pdf

[JNDK10-1] Jarvinen, I., Nyrhinen. A., Ding, A. and M. Kojo, "Effect of IW and Initial RTO changes", Presented to 79th IETF TCPM working group meeting, Nov. 2010. URL http://www.ietf.org/proceedings/79/slides/tcpm-1.pdf

[LAJW07] Liu, D., Allman, M., Jin, S. and L. Wang, "Congestion Control Without a Startup Phase", Protocols for Fast, Long Distance Networks (PFLDnet) Workshop, February 2007. URL http://www.icir.org/mallman/papers/jumpstart-pfldnet07.pdf

[PK98] Padmanabhan V.N. and R. Katz, "TCP Fast Start: A technique for speeding up web transfers", in Proceedings of IEEE Globecom '98 Internet Mini-Conference, 1998.

[PRAKS02] Partridge, C., Rockwell, D., Allman, M., Krishnan, R. and J. Sterbenz, "A Swifter Start for TCP", Technical Report No. 8339, BBN Technologies, March 2002.

[PWSB09] Papadimitriou, D., Welzl, M., Scharf, M. and B. Briscoe, "Open Research Issues in Internet Congestion Control", section 3.4, Internet-draft draft-irtf-iccrg-welzl-congestion-control-open-research-05.txt, work in progress.

[RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.

[RFC2414] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's Initial Window", RFC 2414, September 1998.

[RFC3042] Allman, M., Balakrishnan, H. and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[RFC3150] Dawkins, S., Montenegro, G., Kojo, M. and V. Magret, "End-to-end Performance Implications of Slow Links", RFC 3150,

July 2001.

[RFC3782] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004.

[RFC4782] Floyd, S., Allman, M., Jain, A. and P. Sarolahti, "Quick-Start for TCP and IP", RFC 4782, January 2007.

[RJ10] Ramachandran, S. and A. Jain, "Aggregate Statistics of Size Related Metrics of Web Pages metrics", 2010. URL http://code.google.com/speed/articles/web-metrics.html

[Sch08] Scharf, M., "Quick-Start, Jump-Start, and Other Fast Startup Approaches", November 17, 2008. URL http://www.ietf.org/old/2009/proceedings/08nov/slides/iccrg-2.pdf

[Sch11] Scharf, M., "Performance and Fairness Evaluation of IW10 and Other Fast Startup Schemes", Presented to 80th IRTF ICCRG working group meeting, Nov. 2010. URL http://www.ietf.org/proceedings/80/slides/iccrg-1.pdf

[Sch11-1] Scharf, M., "Comparison of end-to-end and network-supported fast startup congestion control schemes", Computer Networks, Feb. 2011. URL http://dx.doi.org/10.1016/j.comnet.2011.02.002

[SPDY] "SPDY: An experimental protocol for a faster web", URL http://dev.chromium.org/spdy

[Ste08] Sounders S., "Roundup on Parallel Connections", High Performance Web Sites blog. URL http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections

[Tou10] Touch, J., "Automating the Initial Window in TCP", Internet-draft draft-touch-tcpm-automatic-iw-00.txt, work in progress.

[VH97] Visweswaraiah, V. and J. Heidemann, "Improving Restart of Idle TCP Connections", Technical Report 97-661, University of Southern California, November 1997.

Author's Addresses

H.K. Jerry Chu
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA
EMail: hkchu@google.com

Nandita Dukkipati
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA
EMail: nanditad@google.com

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA
EMail: ycheng@google.com

Matt Mathis
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA
EMail: mattmathis@google.com

Acknowledgement

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

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: January 12, 2012

M. Mathis
N. Dukkipati
Y. Cheng
Google, Inc
July 11, 2011

Proportional Rate Reduction for TCP
draft-mathis-tcpm-proportional-rate-reduction-01.txt

Abstract

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) and related algorithms to improve the accuracy of the amount of data sent by TCP during loss recovery. Standard Congestion Control requires that TCP and other protocols reduce their congestion window in response to losses. This window reduction naturally occurs in the same round trip as the data retransmissions to repair the losses, and is implemented by choosing not to transmit any data in response to some ACKs arriving from the receiver. Two widely deployed algorithms are used to implement this window reduction: Fast Recovery and Rate Halving. Both algorithms are needlessly fragile under a number of conditions, particularly when there is a burst of losses that such that the number of ACKs returning to the sender is so small that the effective window falls below the target congestion window chosen by the congestion control algorithm. Proportional Rate Reduction avoids these excess window reductions such that at the end of recovery the actual window size will be as close as possible to the window size determined by the congestion control algorithm. It is patterned after rate halving, but using the fraction that is appropriate for target window chosen by the congestion control algorithm. In addition we propose two slightly different algorithms to bound the total window reduction due to all mechanisms, including application stalls, the losses themselves and inhibit further window reductions when possible.

Status of this Memo

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 12, 2012.

Table of Contents

1. Introduction 4
2. Definitions 5
3. Algorithms 6
3.1. Examples 7
4. Properties 10
5. Measurements 12
6. Conclusion and Recommendations 13
7. Acknowledgements 14
8. Security Considerations 14
9. IANA Considerations 14
10. References 14
Appendix A. Packet Conservation Bound 15
Authors' Addresses 16

1. Introduction

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) and two slightly different reduction bound algorithms to improve the accuracy of the amount of data sent by TCP during loss recovery.

Standard Congestion Control [RFC 5681] requires that TCP (and other protocols) reduce their congestion window in response to losses. Fast Recovery, described in the same document, is the reference algorithm for making this adjustment. It's stated goal is to recover TCP's self clock by relying on returning ACKs during recovery to clock more data into the network. Fast Recovery adjusts the window by waiting for one half RTT of ACKs to pass before sending any data. It is fragile because it can not compensate for the implicit window reduction caused by the losses themselves, and is exposed to timeouts. For example if half of the data or ACKs are lost, Fast Recovery's expected behavior would be to reduce the window by not sending in response to the first half window of ACKs, but then it would not receive any additional ACKs and would timeout because it failed to send anything at all.

The rate-halving algorithm improves this situation by sending data on alternate ACKs during recovery, such that after one RTT the window has been halved. Rate-having is implemented in Linux after only being informally published [RHweb], including from an uncompleted Internet-Draft[RHID]. Rate-halving also does not adequately compensate for the implicit window reduction caused by the losses and also assumes a 50% window reduction, which was completely standard at the time it was written (several modern congestion control algorithms, such as Cubic[CUBIC], can sometimes reduce the window by much less than 50%.). As a consequence rate-halving often allows the window to fall further than necessary, reducing performance and increasing the risk of timeouts if there are any additional losses.

Proportional Rate Reduction (PPR) avoids these excess window reductions such that at the end of recovery the actual window size will be as close as possible to the window size determined by the congestion control algorithm. It is patterned after Rate Halving, but using the fraction that is appropriate for target window chosen by the congestion control algorithm. During PRR one of two additional reduction bound algorithms monitors the total window reduction due to all mechanisms, including application stalls, the losses themselves and attempts to inhibit further window reductions.

We describe two slightly different reduction bound algorithms: conservative reduction bound (CRB), which meets a strict segment conserving correctness critera; and a slow start reduction bound

(SSRB), which is more aggressive than CRB by at most one segment per ACK. CRB meets a conservative, philosophically pure and aesthetically appealing notion of correct, however in real networks it does not perform as well as the algorithms described in RFC 3517, which prove to be non-conservative in a statistically significant number of cases. SSRB offers a compromise by allowing TCP to send one additional segment per ACK relative to CRB in some situations. Although SSRB is less aggressive than RFC 3517 (transmitting fewer segments or transmitting them later) it slightly outperforms it, due to slightly lower probability of additional losses during recovery.

All three algorithms are based on common design principles, derived from Van Jacobson's packet conservation principle: segments delivered to the receiver are used as the clock to trigger sending additional segments into the network. As much as possible Proportional Rate Reduction and the reduction bound rely on this self clock process, and are only slightly affected by the accuracy of other estimators, such as pipe[RFC 3517] and cwnd. This is what gives the algorithms their precision in the presence of events that cause uncertainty in other estimators.

In Section 5, we summarize a companion paper[Recovery] with some measurement experiments: PRR+SSRB outperforms both RFC 3517 and PRR+ CRB under authentic network traffic.

The algorithms are described as modifications to RFC 5681, TCP Congestion Control, using concepts drawn from the pipe algorithm [RFC 3517]. They are most accurate and more easily implemented with SACK[RFC 2018], but they do not require SACK.

2. Definitions

The following terms, parameters and state variables are used as they are defined in earlier documents:

RFC 3517: covered (as in "covered sequence numbers")

RFC 5681: duplicate ACK, FlightSize, Sender Maximum Segment Size (SMSS)

Voluntary Window Reductions: choosing not to send data in response to some ACKs, for the purpose of reducing the sending window size or data rate.

We define some additional variables:

SACKd: The total number of bytes that the scoreboard indicates has

been delivered to the receiver. This can be computed by scanning the scoreboard and counting the total number of bytes covered by all sack blocks.

DeliveredData: The total number of bytes that the current ACK indicates have been delivered to the receiver, relative to all past ACKs. When not in recovery, DeliveredData is the change in snd.una. With SACK, DeliveredData is not an estimator and can be computed precisely as the change in snd.una plus the change in SACKd. Note that if there are SACK blocks and snd.una advances, the change in SACKd is typically negative. In recovery without SACK, DeliveredData is estimated to be 1 SMSS on duplicate acknowledgements, and on a subsequent partial or full ACK, DeliveredData is estimated to be the change in snd.una, minus one SMSS for each preceding duplicate ACK.

Note that DeliveredData is robust: for TCP using SACK, DeliveredData can be precisely computed anywhere in the network just by inspecting the returning ACKs. The consequence of missing ACKs is that later ACKs will show a larger DeliveredData. Furthermore, for any TCP (with or without SACK) the sum of DeliveredData must agree with the forward progress over the same time interval.

We introduce a local variable "sndcnt", which indicates exactly how many bytes should be sent in response to each ACK. Note that the decision of which data to send (e.g. retransmit missing data or send more new data) is out of scope for this document.

3. Algorithms

Summary: If pipe (the estimated data is in flight) is larger than ssthresh (the target cwnd at the end of recovery) then Proportional Rate Reduction spreads the the voluntary window reductions across a full RTT, such that as prr_delivered approaches RecoverFS (at the end of recovery) prr_out approaches ssthresh, the target value for cwnd. If there are excess losses such that pipe falls below ssthresh, the selected reduction bound algorithm tries to hold pipe at ssthresh by sending at most "limit" segments per ACK to catch up. For both reduction bound algorithms the limit is first set to past voluntary window reductions (prr_delivered - prr_out) permitting single ACKs to trigger sending multiple segments. With PRR+CRB (conservative==True) and if there are too many losses then prr_delivered - prr_out will be exactly the same as DeliveredData for the current ACK, resulting in sndcnt=DeliveredData. Therefore there will be no further Voluntary Window Reductions. With PRR+SSRB (conservative==False) the same situation results in sndcnt=DeliveredData+1, which ultimately causes TCP to slowstart up to ssthresh.

At the beginning of recovery initialize PRR state. This assumes a modern congestion control algorithm, CongCtrlAlg(), that might set ssthresh to something other than FlightSize/2:

```
ssthresh = CongCtrlAlg() // Target cwnd after recovery
prr_delivered = 0         // Total bytes delivered during recov
prr_out = 0              // Total bytes sent during recovery
RecoverFS = snd.nxt-snd.una // Flightsize at the start of recov
```

On every ACK during recovery compute:

```
DeliveredData = delta(snd.una) + delta(SACKd)
prr_delivered += DeliveredData
pipe = (RFC 3517 pipe algorithm)
if (pipe > ssthresh) {
   // Proportional Rate Reduction
   sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
} else {
   // Two version of the reduction bound
   if (conservative) {    // PRR+CRB
     limit = prr_delivered - prr_out
   } else {               // PRR+SSRB
     limit = MAX(prr_delivered - prr_out, DeliveredData) + 1
   }
   sndcnt = MIN(ssthresh - pipe, limit)
}
sndcnt = MAX(sndcnt, 0)                    // positive
```

On any data transmission or retransmission:

```
   prr_out += (data sent) // strictly less than or equal to sndcnt
```

The following examples will make these algorithms much clearer.

3.1. Examples

We illustrate these algorithms by showing their different behaviors for two scenarios: TCP experiencing either a single loss or a burst of 15 consecutive losses. In all cases we assume bulk data, standard AIMD congestion control (the ssthresh is set to Flight Size/2) and cwnd = FlightSIze = pipe = 20 segments, so ssthresh will be set to 10 at the beginning of recovery. We also assume standard Fast Retransmit and Limited Transmit, so we send two new segments followed by one retransmit on the first 3 duplicate ACKs after the losses.

Each of the diagrams below shows the per ACK response to the first round trip for the various recovery algorithms when the zeroth segment is lost. The top line indicates the transmitted segment

number triggering the ACKs, with an X for the lost segment. "cwnd" and "pipe" indicate the values of these algorithms after processing each returning ACK. "Sent" indicates how much "N"ew or "R"etransmitted data would be sent. Note that the algorithms for deciding which data should be sent are out of scope of this document.

When there is a single loss, PRR with either of the reduction bound algorithms has the same behavior. We show "RB", a flag indicating which reduction bound subexpression ultimately determined the value of sndcnt. When there is minimal losses "limit" (both algorithms) will always be larger than ssthresh - pipe, so the sndcnt will be ssthresh - pipe indicated by "s" in the "RB" row. PRR does not use cwnd during recovery.

RFC 3517

ack#	X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cwnd:		20	20	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
pipe:		19	19	18	18	17	16	15	14	13	12	11	10	10	10	10	10	10	10	10
sent:		N	N	R									N	N	N	N	N	N	N	N

Rate halving (Linux)

ack#	X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cwnd:		20	20	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11
pipe:		19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10
sent:		N	N	R		N		N		N		N		N		N		N		N

PRR

ack#	X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
pipe:		19	19	18	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	10
sent:		N	N	R		N		N		N		N		N		N			N	N
RB:																			s	s

Note that all three algorithms send same total amount of data. RFC 3517 experiences a "half-window of silence", while the Rate Halving and PRR spread the voluntary window reduction across an entire RTT.

Next we consider the same initial conditions when the first 15 packets (0-14) are lost. During the remainder of the lossy RTT, only 5 ACKs are returned to the sender. We examine each of these algorithms in succession.

```
RFC 3517
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
cwnd:                                              20 20 11 11 11
pipe:                                              19 19  4 10 10
sent:                                               N  N 7R  R  R


Rate Halving (Linux)
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
cwnd:                                              20 20  5  5  5
pipe:                                              19 19  4  4  4
sent:                                               N  N  R  R  R


PRR-CRB
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
pipe:                                              19 19  4  4  4
sent:                                               N  N  R  R  R
RB:                                                       f  f  f


PRR-SSRB
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
pipe:                                              19 19  4  5  6
sent:                                               N  N 2R 2R 2R
RB:                                                       d  d  d
```

In this situation, RFC 3517 is very non-conservative, because once fast retransmit is triggered (on the ACK for segment 17) TCP immediately retransmits sufficient data to bring pipe up to cwnd. Our measurement data (see Section 5) indicates that RFC 3517 significantly outperforms Rate Halving, PRR-CRB and some other similarly conservative algorithms that we tested, suggesting that it is significantly common for the actual losses to exceed the window reduction determined by the congestion control algorithm.

The Linux implementation of Rate Halving includes an early version of the conservative reduction bound[RHweb]. In this situation each of the five ACKs trigger exactly 5 transmissions (2 new data, 3 old data), and cwnd is set to 5. At a window size of 5, it takes three round trips to retransmit 15 lost segments. Rate Halving does not raise the window during recovery, so when recovery finally completes, TCP will slowstart cwnd from 5 up to 10. In this example, TCP operates at half of the window chosen by the congestion control for more than three RTTs, increasing the elapsed time and exposing it to timeouts if there are additional losses.

PRR-CRB implements conservative reduction bound. Since the total losses bring pipe below ssthresh, data is sent such that the total data transmitted, prr_out, follows the total data delivered to the receiver as reported by returning ACKs. Transmission are controlled by the sending limit, which was set to prr_delivered - prr_out. This is indicated by the RB:f tagging in the figure. In this case PRR-CRB is exposed to exactly the same problems as Rate Halving, taking excessively long to recover from the losses and being exposed to additional timeouts.

PRR-SSRB increases the window by exactly 1 segment per ACK until pipe rises to sshthresh during recovery. This is accomplished by setting limit to one greater than the data reported to have been delivered to the receiver on this ACK, effectively implementing a slowstart during recovery, and indicated by RB:d tagging in the figure. Although increasing the window during recovery seems to be ill advised, it is important to remember that this actually less aggressive than the current standard which permits sending the same quantity of extra data as a single burst in response to the ACK that triggered Fast Retransmit

Under less extreme conditions, when the total losses are smaller than the difference between Flight Size and ssthresh, PRR-CRB and PRR-SSRB have identical behaviours.

4. Properties

The following properties are common to both PRR-CRB and PRR-SSRB:

Normally Proportional Rate Reduction will spread Voluntary Window reductions out evenly across a full RTT. This has the potential to generally reduce the burstiness of Internet traffic, and could be considered to be a type of soft pacing. Theoretically any pacing increases the probability that different flows are interleaved, reducing the opportunity for ACK compression and other phenomena that increase traffic burstiness. However these effects have not been quantified.

If there are minimal losses, Proportional Rate Reduction will converge to exactly the target window chosen by the congestion control algorithm. Note that as TCP approaches the end of recovery prr_delivered will approach RecoverFS and sndcnt will be computed such that prr_out approaches ssthresh.

Implicit window reductions due to multiple isolated losses during recovery cause later Voluntary Reductions to be skipped. For small numbers of losses the window size ends at exactly the window chosen

by the congestion control algorithm.

For burst losses, earlier Voluntary Window Reductions can be undone by sending extra segments in response to ACKs arriving later during recovery. Note that as long as some Voluntary Window Reductions are not undone, the final value for pipe will be the same as ssthresh, the target cwnd value chosen by the congestion control algorithm.

Proportional Rate Reduction with either reduction round improves the situation when there are application stalls (e.g. when the sending application does not queue data for transmission quickly enough or the receiver stops advancing rwnd). When there is a application stall early during recovery prr_out will fall behind the sum of the transmissions permitted by sndcnt. The missed opportunities to send due to stalls are treated like banked Voluntary Window Reductions: specifically they cause prr_delivered-prr_out to be significantly positive. If the application catches up while TCP is still in recovery, TCP will send a partial window burst to catch up to exactly where it would have been, had the application never stalled. Although this burst might be viewed as being hard on the network, this is exactly what happens every time there is a partial RTT application stall while not in recovery. We have made the partial RTT stall behavior uniform in all states. Changing this behavior is out of scope for this document.

Proportional Rate Reduction with Reduction Bound is significantly less sensitive to errors of the pipe estimator. While in recovery, pipe is intrinsically an estimator, using incomplete information to guess if un-SACKed segments are actually lost or out-of-order in the network. Under some conditions pipe can have significant errors, for example when a burst of reordered data is presumed to be lost and is retransmitted, but then the original data arrives before the retransmission. If the transmissions are regulated directly by pipe as they are in RFC 3517, then errors and discontinuities in the value of the pipe estimator can cause significant errors in the amount of data sent. With Proportional Rate Reduction with Reduction Bound, pipe merely determines how sndcnt is computed from DataDelivered. Since short term errors in pipe are smoothed out across multiple ACKs and both Proportional Rate Reduction and the reduction converge to the same final window, errors in the pipe estimator have less impact on the final outcome (This needs to be tested better).

Under all conditions and sequences of events during recovery, PRR-CRB strictly bounds the data transmitted to be equal to or less than the amount of data delivered to the receiver. We claim that this packet conservation bound is the most aggressive algorithm that does not lead to additional forced losses in some environments. It has the property that if there is a standing queue at a bottleneck that is

carrying no other traffic, the queue will maintain exactly constant length for the entire recovery duration (except for +1/-1 fluctuation due to differences in packet arrival and exit times) . See Appendix A for a detailed discussion of this property.

Although the packet Packet Conserving Bound in very appealing for a number of reasons, our measurements summarized in Section 5 demonstrate that it is less aggressive and does not perform as well as RFC3517, which permits large bursts of data when there are bursts of losses. PRR-SSRB is a compromise that permits TCP to send one extra segment per ACK as compared to the packet conserving bound. From the< perspective of the packet conserving bound, PRR-SSRB does indeed open the window during recovery, however it is significantly less aggressive than RFC3517 in the presence of burst losses.

5. Measurements

In a (to be published) companion paper[Recovery] we describe some measurements comparing the various strategies for reducing the window during recovery. The results presented in that paper are summarized here.

The various window reduction algorithms and extensive instrumentation were all implemented in a modified Linux 2.6.34 kernel. For all experiments we used a uniform subset of the non-standard algorithms present in the base Linux implementation. Specifically we disabled threshold retransmit [FACK], which triggers Fast Retransmit earlier than the standard algorithm. We left enabled CUBIC [CUBIC], limited transmit [LT], and lost retransmission detection algorithms. This subset was a compromise chosen such that the behaviors of both Rate Halving (the Linux default) and RFC 3517 mode were authentic to their respective specifications while at the same time the performance and features were comparable to the kernels in production use. The different window reduction algorithms were all present in the same kernel and could be selected with a sysctl, such that we had an absolutely uniform baseline for comparing RFC 3517, Rate Halving, and PRR with various reduction bounds.

Our experiments included an additional algorithm, PRR with an unlimited bound (PRR-UB), which sends ssthresh-pipe bursts when pipe falls below ssthresh. This behavior parallels RFC 3517.

An important detail of this configuration is that CUBIC only reduces the window by %, as opposed to the 50% reduction used by traditional congestion control algorithms. This, in conjunction with using only standard algorithms to trigger Fast Retransmit, accentuates the tendency for RFC 3517 and PRR-UB to send a burst at the point when

Fast Retransmit gets triggered if pipe is already below ssthresh.

All experiments were performed on servers carrying production traffic for multiple Google services.

In this configuration is is observed that for 32% of the recovery events, pipe falls below ssthresh before Fast Retransmit is triggered, thus the various PRR algorithms start in the reduction bound phase, and both PRR-UB and RFC 3517 send bursts of segments with the fast retransmit.

In the companion paper we observe that PRR-SSRB spends the least time in recovery of all the algorithms tested, largely because it experiences fewer timeouts once it is already in recovery.

RFC 3517 experiences 29% more detected lost retransmissions and 2.6% more timeouts (presumably due to undetected lost retransmissions) than PRR-SSRB. These results are representative of PRR-UB and other algorithms that send bursts when pipe falls below ssthresh.

Rate Halving experiences 5% more timeouts and significantly smaller final cwnd values at the end of recovery. The smaller cwnd sometimes causes the recovery itself to take extra round trips. These results are representative of PRR-CRB and other algorithms that implement strict packet conservation during recovery.

6. Conclusion and Recommendations

[This text assumes standards track. Experimental status would be somewhat more reserved.]

Although the packet conserving bound in very appealing for a number of reasons, our measurements summarized in Section 5 demonstrate that it is less aggressive and does not perform as well as RFC3517, which permits large bursts of data when there are bursts of losses. PRR-SSRB is a compromise that permits TCP to send one extra segment per ACK as compared to the packet conserving bound. From the perspective of the packet conserving bound, PRR-SSRB does indeed open the window during recovery, however it is significantly less aggressive than RFC3517 in the presence of burst losses.

All TCP implementations SHOULD implement both PRR-CRB and PRR-SSRB, with a control to select which algorithm is used. It is RECOMMENDED that PRR-SSRB is the default algorithm.

7. Acknowledgements

This draft is based in part on previous incomplete work by Matt Mathis, Jeff Semke and Jamshid Mahdavi[RHID] and influenced by several discussion with John Heffner.

Monia Ghobadi and Sivasankar Radhakrishnan helped analyze the experiments.

8. Security Considerations

Proportional Rate Reduction does not change the risk profile for TCP.

Implementers that change PRR from counting bytes to segments have to be cautious about the effects of ACK splitting attacks[SPLIT], where the receiver acknowledges partial segments for the purpose of confusing the sender's congestion accounting.

9. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

10. References

TODO: A proper reference section.

[RFC 3517] "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP". E. Blanton, M. Allman, K. Fall, L. Wang. April 2003.

[RFC 5681] "TCP Congestion Control". M. Allman, V. Paxson, E. Blanton. September 2009.

[RHweb] "TCP Rate-Halving with Bounding Parameters". M. Mathis, J. Madavi, http://www.psc.edu/networking/papers/FACKnotes/971219/, Dec 1997.

[RHID] "The Rate-Halving Algorithm for TCP Congestion Control". M. Mathis, J. Semke, J. Mahdavi, K. Lahey. http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt, Work in progress, last updated June 1999.

[CUBIC] "CUBIC: A new TCP-friendly high-speed TCP variant". I. Rhee, L. Xu, PFLDnet, Feb 2005.

[FACK] M. Mathis, J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control", Proceedings of SIGCOMM'96, August, 1996, Stanford, CA.

[Recovery] N. Dukkipati, M. Mathis, Y Cheng, "Improving TCP loss recovery", to be published 2011.

Appendix A. Packet Conservation Bound

Under all conditions and sequences of events during recovery, PRR-CRB strictly bounds the data transmitted to be equal to or less than the amount of data delivered to the receiver. We claim that this packet conservation bound is the most aggressive algorithm that does not lead to additional forced losses in some environments. It has the property that if there is a standing queue at a bottleneck that is carrying no other traffic, the queue will maintain exactly constant length for the entire recovery duration (except for +1/-1 fluctuation due to differences in packet arrival and exit times) . Any less aggressive algorithm will result in a declining queue at the bottleneck. Any more aggressive algorithm will result in an increasing queue or additional losses at the bottleneck.

We demonstrate this property with a little thought experiment:

Imagine a network path that has insignificant delays in both directions, except the processing time and queue at a single bottleneck in the forward path. By insignificant delay, I mean when a packet is "served" at the head of the bottleneck queue, the following events happen in much less than one packet time at the bottleneck: the packet arrives at the receiver; the receiver sends an ACK; which arrives at the sender; the sender processes the ACK and sends some data; the data is queued at the bottleneck.

If sndcnt is set to DataDelivered and nothing else is inhibiting sending data, then clearly the data arriving at the bottleneck queue will exactly replace the data that was served at the head of the queue, so the queue will have a constant length. If queue is drop tail and full then the queue will stay exactly full, even in the presence of losses or reordering on the ACK path, and independent of whether the data is in order or out-of-order (e.g. simple reordering or loss recovery from an earlier RTT). Any more aggressive algorithm sending additional data will cause a queue overflow and loss. Any less aggressive algorithm will under fill the queue. Therefore setting sndcnt to DataDeliverd is the most aggressive algorithm that

does not cause forced losses in this simple network. Relaxing the assumptions (e.g. making delays more authentic and adding more flows, delayed ACKs, etc) increases the noise (jitter) in the system but does not change it's basic behavior.

Note that the congestion control algorithm implements a broader notion of optimal that includes appropriately sharing of the network. PRR-CRB will choose to send the lessor of the data permitted by this packet conserving bound and as determined by the congestion control algorithm as PRR brings TCP's actual window down to ssthresh.

Authors' Addresses

Matt Mathis
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: mattmathis@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: nanditad@google.com

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: ycheng@google.com

TCPM Working Group
Internet Draft
Intended status: Standards Track
Expires: January 2012

J. Touch
USC/ISI

July 7, 2011

Automating the Initial Window in TCP
draft-touch-tcpm-automatic-iw-01.txt

Status of this Memo

Copyright Notice

Abstract

The Initial Window (IW) provides the starting point for TCP's feedback-based congestion control algorithm. Its value has increased over time to increase performance and to reflect increased capability of Internet devices. This document describes a mechanism to adjust the IW over long timescales, to make future changes more safely deployed and to potentially avoid reexamination of this value in the future.

Table of Contents

1. Introduction....2
2. Conventions used in this document....3
3. Design Considerations....3
4. Proposed IW Algorithm....4
5. Discussion....6
6. Security Considerations....8
7. IANA Considerations....9
8. Conclusions....9
9. References....9
   9.1. Normative References....9
   9.2. Informative References....10
10. Acknowledgments....10

1. Introduction

TCP's congestion control algorithm uses an initial window value (IW), both as a starting point for new connections and after one RTO or more [RFC2581][RFC2861]. This value has evolved over time, originally one maximum segment size (MSS), and increased to the lesser of four MSS or 4,380 bytes [RFC3390][RFC5681]. For typical Internet connections with an maximum transmission units (MTUs) of 1500 bytes, this permits three segments of 1,460 bytes each.

The IW value was originally implied in the original TCP congestion control description, and documented as a standard in 1997 [RFC2001][Ja88]. The value was last updated in 1998 experimentally, and moved to the standards track in 2002 [RFC2414][RFC3390]. There have been recent proposals to update the IW based on further increases in host and router capabilities and network capacity, some focusing on specific values (e.g., IW=10), and others prescribing a schedule for increases over time (e.g., IW=6 for 2011, increasing by 1-2 MSS per year).

This document proposes that TCP can objectively measure when an IW is too large, and that such feedback should be used over long timescales to adjust the IW automatically. The result should be safer to deploy and might avoid the need to repeatedly revisit IW size over time.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

In this document, the characters ">>" preceding an indented line(s) indicates a compliance requirement statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the explicit compliance requirements of this RFC.

3. Design Considerations

TCP's IW value has existed statically for over two decades, so any solution to adjusting the IW dynamically should have similarly stable, non-invasive effects on the performance and complexity of TCP. In order to be fair, the IW should be similar for most machines on the public Internet. Finally, a desirable goal is to develop a self-correcting algorithm, so that IW values that cause network problems can be avoided. To that end, we propose the following list of design goals:

- Little to no impact to TCP in the absence of loss, i.e., it should not increase the complexity of default packet processing in the normal case.

- Adapt to network feedback over long timescales, avoiding values that persistently cause network problems.

We expect that, without other context, a good IW algorithm will converge to a single value, but this is not required. An endpoint with additional context or information, or deployed in a constrained environment, can always use a different value. In specific, information from previous connections, or sets of connections with a similar path, can already be used as context for such decisions [RFC2140].

However, if a given IW value persistently causes packet loss during the initial burst of packets, it is clearly inappropriate and could be inducing unnecessary loss in other competing connections. This might happen for sites behind very slow boxes with small buffers, which may or may not be the first hop.

4. Proposed IW Algorithm

Below is a simple description of the proposed IW algorithm. It relies on the following parameters:

- MinIW = 3 MSS or 4,380 bytes (as per RFC3390]
- MaxIW = date.year - 2000
- MulDecr = 0.5
- AddIncr = 2 MSS
- Threshold = 0.05

We assume that the minimum IW (MinIW) should be as currently specified [RFC3390]. The maximum IW can either be set to a fixed value [Ch10], or set based on a schedule [Al10]. Regardless, we propose that the value adapt over time, so have specified it in terms of the current date. If that is not feasible or the time is not available, a fixed value can be used. We also propose to use an AIMD algorithm, with increase and decreases as noted.

Note that all of these parameters are up for discussion, though should be determined by the time this document is issued as an RFC. We do not anticipate that any of them are critical to the overall design, especially because both current proposals are degenerate cases of the algorithm below for given parameters (notably MulDec = 1.0 and AddIncr = 0 MSS, thus disabling the automatic part of the algorithm).

The specific algorithm is as follows:

0. On boot:

```
IW = MaxIW; # assume this is in bytes, and an even number of MSS
```

1. Upon starting a new connection

```
CWND = IW;
conncount++;
IWnotchecked = 1; # true
```

2. If SYN-ACK includes ECN, treat as if the IW is too large

```
if (synackecn == 1) {
   losscount++;
   IWnotchecked = 0; # never check again
}
```

3. During a retransmission, check the seqno of the outgoing packet (in bytes)

```
if (IWnotchecked && ((ISN - seqno) < IW))) {
   losscount++;
   IWnotchecked = 0; # never do this entire "if" again
} else {
   IWnotchecked = 0; # you're beyond the IW so stop checking
}
```

4. Once a month or once every 1000 connections if no date is available:

```
if ((monthly == TRUE) || (conncount > 1000)) {
   if (losscount/conncount > threshold) {
      # the number of connections with errors is too high
      IW = IW * MulDecr;
   } else {
      IW = IW + AddIncr;
   }
}
```

We recognize that this algorithm can yield a false positive when the sequence number wraps around. In that case, we might be able to use PAWS to avoid the issue, encourage the use of 64-bit sequence numbers internal to the implementation, or ignore the issue and just allow the false positives [RFC1323].

Standards language (as a shopping list):

MAY implement this as an alternative to RFC3390

If implemented:

MUST start IW at MaxIW - i.e., IW in the absence of other info

MUST limit MaxIW growth (Static or to year if poss)

MUST check once a month or 1,000 connections (the larger)

MUST decrease by at least 0.5x

MUST NOT increase by more than 2 MSS

SHOULD use IW that is integer multiple of 2 MSS (for ACK compression)

MUST decrease IW if > 95% connections have errors

MAY increase IW otherwise

But MUST limit increase to 2 MSS/year (is this needed?)

SHOULD be implemented to limit performance impact

SHOULD be implemented to avoid seqno wrap issues

(anything else?)

There are some TCP connections which might not be counted at all, such as those to/from loopback addresses, or those within the same subnet as that of a local interface (for which congestion control is sometimes disabled anyway). This may also include connections that terminate before the IW is full, i.e., as a separate check at the time of the connection closing.

The period over which the IW is updated is intended to be a long timescale, e.g., a month or so, or 1,000 connections, whichever is longer. An implementation might check the IW once a month, and simply not update the IW or clear the connection counts in months where the number of connections is too small.

5. Discussion

The following is intended as a list of notes to be discussed:

- Algorithm uses IW as an even multiple of MSS due to ACK compression
- Impact of SEQNO wraparound vs. use of PAWS
- Algorithm now assumes bytes, not segments

- Algorithm now counts losses only during the first IW after start; should the system ignore rechecking the burst after idle, i.e., do checks only once on the initial connection? To fix this, step #2 would use "IWstart" as the front of the IW, set this to ISN at connection start, and reset it to seqno during a slow-start restart. This isn't a lot of code, and takes effect only during restarts anyway - it's not in the fast path either.

- Impact of spurious retransmissions due to reordering (false positive)

- Granularity (per-machine -same as now, per-interface? per-subnet? vs. cost?)

- Need to keep ISN - needed for other uses (e.g., TCP-AO), and typically kept except in Linux.

- Need for persistent state if a reboot occurs within the 1-month window of evaluation

- Degenerate case due to failure is to act as if a fixed window, which is what we have now

- Interaction with 2140

Basically 2140 sets CWND to something other than IW when it knows better; this doc is for IW which is used there only for 'new' places (or forgotten old ones).

- Explain why RWIN is not involved

  Receiver-limited space

  Space for reordering

  NOT congestion control

  Although sender window isn't useful if larger than this

  CWND is a path property; RWIN is an endpoint property

- Reasons not to report-back:

- privacy concerns

- opportunity for spoofer poisoning the data (more on that in the doc)

- using a DNS query is a bad idea
  - requires every TCP stack support DNS queries
  - requires a resolution step in addition to the reporting
  - could cause the kernel to block on a timeout

- biased reporting
  if cellphones (e.g.) never do this, we won't know about a potentially large percent of endpoints

6. Observations

- The IW may not converge to a global value; that's OK.

- IW values can fluctuate; there should not be a significant impact to this if that's what's seen by this algorithm.

- We do assume that losses during the IW are due to the IW being too large; persistent errors that drop packets for other reasons (e.g., OS bugs) can cause false positives, however this is consistent with TCP's general assumption that loss is caused by congestion that requires backoff. This algorithm treats the IW of new connections as a long-timescale backoff system.

7. Security Considerations

Obvious ones - poisoning the info (fake loss, fake success), what happens when one party disobeys, and whether anything is different

---

You can already do that within a connection too. Yes, you can pollute aggregate info by virtue of it being aggregate. There's a tradeoff of trust here - how much do you believe what's happening most of the time, and how do you react to it.

If most of the connections lie about receiving data, then you see a world where larger IW is working, and unless you detect data loss some other way, TCP worked exactly as it should.

IMO, the good news is that:

- the IW drops if you get lots of lies about dropped packets but then those endpoints could have just dropped the packets, and dropping the IW is the right response anyway

- the IW increases if you get lots of lies about non-drops, but then if you don't see anything else, you have no reason to claim that anything is amiss anyway

So yes, there's spoofing in the aggregate. The law of large numbers - connecting to lots of places - should help reduce that effect. But ultimately it's not all that clear that the reactions of this sort of poisoning aren't the right ones anyway.

In the end, it might be safer to require a high percent of connections react badly to IW (i.e., over 95%) - that means that

a) if you did see loss, someone bad is controlling nearly all of your connections anyway

b) if you don't see loss through TCP, and you didn't detect data drops by other means for that many connections, you really don't have a problem

I.e., increasing the threshold increases your ability to detect the false IW increase case, so it's safer...

8. IANA Considerations

This document has no IANA considerations. This section should be removed prior to publication.

9. Conclusions

<Add any conclusions>

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3390] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window", RFC 3390 (Standards Track), Oct. 2002.

[RFC5681] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5681 (Standards Track), Sep. 2009.

10.2. Informative References

[Al10] Allman, M., "Initial Congestion Window Specification", (work in progress), draft-allman-tcpm-bump-initcwnd-00, Nov. 2010.

[Ch10] Chu, J., Dukkipati, N., Cheng, Y., Mathis, M., "Increasing TCP's Initial Window," (work in progress), draft-ietf-tcpm-initcwnd-01, Apr. 2011.

[Ja88] Jacobson, V., M. Karels, "Congestion Avoidance and Control", Proc. Sigcomm 1988.

[RFC1323] Jacobson, V., Braden, R., Borman, D., "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC2001] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC2001 (Standards Track), Jan. 1997.

[RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140 / STD 7(Informational), Apr. 1997.

[RFC2414] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window", RFC 2414 (Experimental), Sept. 1998.

[RFC2581] Allman, M., Paxson, V., Stevens, W., "TCP Congestion Control," RFC2581 (Standards Track), Apr. 1999.

[RFC2861] Handley, M., Padhye, J., Floyd, S., "TCP Congestion Window Validation", RFC2861 (Experimental),

11. Acknowledgments

Mark Allman and Aki Nyrjinen contributed to the development of this algorithm. Members of the TCPM mailing list also participated in providing useful feedback.

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695 U.S.A.

Phone: +1 (310) 448-9151
Email: touch@isi.edu