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 in August, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Abstract

TCP Fast Open (TFO) allows data to be carried in the SYN and 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 [CDCM11].

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
Internet Draft              TCP Fast Open             February 2012

constraints. E.g., the Chrome browser keeps TCP connections idle up to 5 minutes but 35% of Chrome HTTP requests are made on new TCP connections. More discussions on HTTP persistent connections are in section 7.1.

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 restrict 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 3WS 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 3WS:

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:

<table>
<thead>
<tr>
<th>TCP A (Client)</th>
<th>TCP B (Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>LISTEN</td>
</tr>
<tr>
<td>#1 SYN-SENT</td>
<td>&lt;SYN, CookieOpt=NIL&gt;</td>
</tr>
<tr>
<td>#2 ESTABLISHED</td>
<td>&lt;SYN, ACK, CookieOpt=C&gt;</td>
</tr>
<tr>
<td></td>
<td>(caches cookie C)</td>
</tr>
</tbody>
</table>

Performing TCP Fast Open in connection 2:

<table>
<thead>
<tr>
<th>TCP A (Client)</th>
<th>TCP B (Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>LISTEN</td>
</tr>
<tr>
<td>#1 SYN-SENT</td>
<td>&lt;SYN=x, CookieOpt=C, DATA_A&gt;</td>
</tr>
<tr>
<td>#2 ESTABLISHED</td>
<td>&lt;SYN=y, ACK=x+len(DATA_A)+1&gt;</td>
</tr>
<tr>
<td>#3 ESTABLISHED</td>
<td>&lt;ACK=x+len(DATA_A)+1, DATA_B&gt;</td>
</tr>
<tr>
<td>#4 ESTABLISHED</td>
<td>&lt;ACK=y+1&gt;</td>
</tr>
<tr>
<td>#5 ESTABLISHED</td>
<td>&lt;ACK=y+len(DATA_B)+1&gt;</td>
</tr>
</tbody>
</table>
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 | Length |                              |
| +-----------------------------------------------+ |
|                                               |
|                                               |
| +-----------------------------------------------+ |
| |                                               |
| +-----------------------------------------------+ |
|                                               |
| +-----------------------------------------------+ |
| | Kind | Length |                              |
| +-----------------------------------------------+ |

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

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 | Length |                              |
| +-----------------------------------------------+ |

<table>
<thead>
<tr>
<th>Kind</th>
<th>1 byte: constant TBD (assigned by IANA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>1 byte: range 6 to 18 (bytes); limited by remaining space in the options field. The number MUST be even.</td>
</tr>
<tr>
<td>Cookie</td>
<td>4 to 16 bytes (Length - 2)</td>
</tr>
</tbody>
</table>
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 accept more than one valid cookie per client at any given time. But this is all server implementation dependent and transparent to the client.

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 [RCCJB11].

Note that if only one valid cookie is allowed per-client and the server can regenerate the cookie independently, the best validation
process may be for the server to simply regenerate a valid cookie and compare it against the incoming cookie. In that case if the incoming cookie fails the check, a valid cookie is readily available to be sent to the client without additional computation.

Also note the server may want to use special cookie values, e.g., "0", for specific scenarios. For example, the server wants to notify the client the support of TFO, but chooses not to return a valid cookie for security or performance reasons upon receiving a TFO request.

4.1.3. Client Cookie Handling

The client MUST cache cookies from 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]. The client SHOULD update the cache MSS value whenever it discovers new MSS value, e.g., through path MTU discovery.

Caching RTT allows seeding a more accurate SYN timeout than the default value [RFC6298]. 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.

Note that before TFO sees wide deployment, clients are advised to also cache negative responses from servers in order to reduce the amount of futile TFO attempts. Since TFO is enabled on a per-service port basis but cookies are independent of service ports, clients’ cache should include remote port numbers too.

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 a preset system limit, the server SHOULD disable TFO for all new connection requests until PendingFastOpenRequests drops below the system limit. 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 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 (although one can argue the delay in the data transmission till after 3WHS is justified if the SYN drop is due to network congestion).
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 [RFC6298].

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. Initialize and reset a local FastOpened flag. If FastOpenEnabled is false, go to step 5.

2. If PendingFastOpenRequests is over the system limit, go to step 5.

3. If IsCookieValid() in section 4.1.2 returns false, go to step 5.

4. Buffer the data and notify the application. Set FastOpened flag and increment PendingFastOpenRequests.

5. Send the SYN-ACK packet. The packet MAY include a Fast Open Option. If FastOpened flag is set, the packet acknowledges the SYN and data sequence. Otherwise it acknowledges only the SYN sequence.
The server MAY include data in the SYN-ACK packet if the response data is readily available. Some application may favor delaying the SYN-ACK, allowing the application to process the request in order to produce a response, but this is left to the implementation.

6. Advance to the SYN-RCVD state. If the FastOpened flag is set, 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].

If the SYN-ACK timer fires, the server SHOULD retransmit a SYN-ACK segment with neither data nor 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 can retransmit it in the first ACK packet in step 2. 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 dialup 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 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 not to respond with data until handshake finishes. 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. Web Performance

7.1. HTTP persistent connection

TCP connection setup overhead has long been identified as a performance bottleneck for web applications [THK98]. HTTP persistent connection was proposed to mitigate this issue and has been widely deployed. However, [RCCJR11][AERG11] show that the average number of transactions per connection is between 2 and 4, based on large-scale measurements from both servers and clients. In these studies, the servers and clients both kept the idle connections up to several minutes, well into the human think time.

Can the utilization rate increase by keeping connections even longer? Unfortunately, this is problematic due to middle-boxes and rapidly growing mobile end hosts. One major issue is NAT. Studies [HNESSK10][MQXMZ11] show that the majority of home routers and ISPs fail to meet the the 124 minutes idle timeout mandated in [RFC5382]. In [MQXMZ11], 35% of mobile ISPs timeout idle connections within 30 minutes. NAT boxes do not possess a reliable mechanism to notify endhosts when idle connections are removed from local tables, either due to resource constraints such as mapping table size, memory, or
lookup overhead, or due to the limited port number and IP address space. Moreover, unmapped packets received by NAT boxes are often dropped silently. (TCP RST is not required by RFC5382.) The end host attempting to use these broken connections are often forced to wait for a lengthy TCP timeout. Thus the browser risks large performance penalty when keeping idle connections open. To circumvent this problem, some applications send frequent TCP keep-alive probes. However, this technique drains power on mobile devices [MQXMZ11]. In fact, power has become a prominent issue in modern LTE devices that mobile browsers close the HTTP connections within seconds or even immediately [SOUDERS11].

Idle connections also consume more memory resources. Due to the complexity of today’s web applications, the application layer often needs orders of magnitude more memory than the TCP connection footprint. As a result, servers need to implement advanced resource management in order to support a large number of idle connections.

7.2 Case Study: Chrome Browser

[RCCJR11] studied Chrome browser performance based on 28 days of global statistics. Chrome browser keeps idle HTTP persistent connections up to 5 to 10 minutes. However the average number of the transactions per connection is only 3.3. Due to the low utilization, TCP 3WHS accounts up to 25% of the HTTP transaction network latency. The authors tested a Linux TFO implementation with TFO enabled Chrome browser on popular web sites in emulated environments such as residential broadband and mobile networks. They showed that TFO improves page load time by 10% to 40%. More detailed on the design tradeoffs and measurement can be found at [RCCJB11].

8. TFO’s Applicability

TFO aims at latency conscious applications that are sensitive to TCP’s initial connection setup delay. These application protocols often employ short-lived TCP connections, or employ long-lived connections but are more sensitive to the connection setup delay due to, e.g., a more strict connection failover requirement.

Only transaction-type applications where RTT constitutes a significant portion of the total end-to-end latency will likely benefit from TFO. Moreover, the client request must fit in the SYN packet. Otherwise there may not be any saving in the total number of round trips required to complete a transaction.

To the extent possible applications protocols SHOULD employ long-lived connections to best take advantage of TCP’s built-in congestion control algorithm, and to reduce the impact from TCP’s connection
setup overhead. E.g., for the web applications, P-HTTP will likely help and is much easier to deploy hence should be attempted first. TFO will likely provide further latency reduction on top of P-HTTP. But the additional benefit will depend on how much persistency one can get from HTTP in a given operating environment.

One alternative to short-lived TCP connection might be UDP, which is connectionless hence doesn’t inflict any connection setup delay, and is best suited for application protocols that are transactional. Practical deployment issues such as middlebox and/or firewall traversal may severely limit the use of UDP based application protocols though.

Note that when the application employs too many short-lived connections, it may negatively impact network stability, as these connections often exit before TCP’s congestion control algorithm kicks in. Implementations supporting large number of short-lived connections should employ temporal sharing of TCB data as described in [RFC2140].

More discussion on TCP Fast Open and its projected performance benefit can be found in [RCCJB11].

9. Related Work

9.1. 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.

9.2. Common Defenses Against SYN Flood Attacks

TFO is still vulnerable to SYN flood attacks just like normal TCP

Cheng, et. al.
Expires August 2012
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.

9.3. 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.

Rapid-Restart [SIMPSON11] is based on TCPCT and shares similar goal as TFO. In Rapid-Restart, both the server and the client retain the TCP control blocks after a connection is terminated in order to allow/resume data exchange in next connection handshake. In contrary, TFO does not require keeping both TCB on both sides and is more scalable.

10. 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.

11. Acknowledgements

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

12.1. Normative References


12.2. Informative References


Author’s Addresses

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

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@cs.ucsd.edu

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.
Increasing TCP’s Initial Window

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 August, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
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 usage and deployment recommended by the IETF TCP Maintenance and Minor Extensions (TCPM) working group.

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 ........................................... 3
2. TCP Modification .................................... 4
3. Implementation Issues ................................. 5
4. Background ........................................... 5
5. Advantages of Larger Initial Windows ............... 7
   5.1 Reducing Latency ................................... 7
   5.2 Keeping up with the growth of web object size .... 7
   5.3 Recovering faster from loss on under-utilized or wireless links . 8
7. Disadvantages of Larger Initial Windows for the Network . 9
8. Mitigation of Negative Impact ........................ 10
9. Interactions with the Retransmission Timer .......... 10
10. Experimental Results From Large Scale Cluster Tests .... 10
   10.1 The benefits .................................... 10
   10.2 The cost ....................................... 11
11. Other Studies ....................................... 12
12. Usage and Deployment Recommendations ............... 13
13. Related Proposals .................................. 13
14. Security Considerations ............................. 14
15. Conclusion .......................................... 14
16. IANA Considerations ................................. 14
17. Acknowledgments .................................... 14
18. Normative References ............................... 15
19. Informative References ............................. 15
Appendix A - List of Concerns and Corresponding Test Results .... 19
Author’s Addresses ..................................... 22
Acknowledgment ........................................ 22
1. Introduction

This document proposes to update 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. Due to lingering concerns about possible side effects, some of the recommendations are conditional on additional monitoring and evaluation.

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 recommend that all TCP implementations have a settable TCP IW parameter. As of 2011 an appropriate value for this parameter is 10 segments as long as there is a reasonable effort to monitor for possible interactions with other Internet applications and services. Furthermore, it is understood that the appropriate value for IW is likely to continue to rise in the future, but this document does not include any supporting evidence for larger values of IW.

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 the consensus from the TCPM working group on the near-term usage and deployment of IW10 in the Internet.

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 depending on the MSS. This increase is optional: a TCP MAY start with an initial window that is smaller than 10 segments.

More precisely, the upper bound for the initial window will be

\[
\min (10\times \text{MSS}, \max (2\times \text{MSS}, 14600))
\]

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 during or 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.

Note that all the test results described in this document were based on the regular Ethernet MTU of 1500 bytes. Future study of the effect of a different MTU may be needed to fully validate (1) above.

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 [RFC6298] 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, and more than 4KB of data is sent.

3. Implementation Issues

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 outperforms 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 [RFC6077, 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 [Bell10] 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 above issues with HTTP [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 shows significant latency reduction with the large initial window even in conjunction with these two HTTP features ([Duk10]).

Also note that packet pacing has been suggested as a possible mechanism to avoid large bursts and their associated harm [VH97]. Pacing is not required in this proposal due to a 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.

<table>
<thead>
<tr>
<th>total segments</th>
<th>IW=3</th>
<th>IW=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>46</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>51</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>78</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>79</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>120</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>127</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

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 [All100].
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 such as SPDY [SPDY] allow multiple web objects to be sent in a single transaction, potentially benefiting from an even larger initial window in order to transfer an entire web page in a small number of round trips.

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.

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 the large scale experiments described later.

Until this proposal is widely deployed, a fairness issue may exist between flows adopting a larger initial window vs flows that are RFC3390-compliant. Although no severe unfairness has been detected on all the known tests so far, further study on this topic may be warranted.

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 or more) 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 [Get11-1]. 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.

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

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 6298 [RFC6298] 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. Other Studies

Besides the large scale Internet experiments described above, a number of other studies have been conducted on the effects of IW10 in various environments. These tests were summarized below, with more discussion in Appendix A.

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

1. [Sch08] described an earlier evaluation of various Fast Startup approaches, including the "Initial-Start" of 10 MSS.

2. [DCCM10] presented the result from Google’s large scale IW10 experiments, with a focus on areas with highly multiplexed links or limited broadband deployment such as Africa and South America.

3. [CW10] contained a testbed study on IW10 performance over slow links. It also studied how short flows with a larger initial window might affect the throughput performance of other co-existing, long lived, bulk data transfers.
4. [Sch11] compared IW10 against a number of other fast startup schemes, and concluded that IW10 works rather well and is also quite fair.

5. [JNDK10] and later [JNDK10-1] studied the effect of IW10 over cellular networks.

6. [AERG11] studied the effect of larger ICW sizes, among other things, on end users’ page load time from Yahoo!’s Content Delivery Network.

12. Usage and Deployment Recommendations

Further experiments are required before a larger initial window shall be enabled by default in the Internet. The existing measurement results indicate that this does not cause significant harm to other traffic. However, widespread use in the Internet could reveal issues not known yet, e.g., regarding fairness or impact on latency-sensitive traffic such as VoIP.

Therefore, special care is needed when using this experimental TCP extension, in particular on large-scale systems originating a significant amount of Internet traffic. Anyone (stack vendors, network administrators, etc.) turning on a larger initial window SHOULD ensure that the performance is monitored before and after that change. Relevant performance metrics include the percentages of packet losses or segment retransmissions as well as application-level metrics such as data transfer completion times or media quality. Note that it is important also to take into account hosts that do not implement a larger initial window. Furthermore, non-TCP traffic (such as VoIP) should be monitored as well. If users observe any significant deterioration of performance, they SHOULD fall back to an initial window as allowed by RFC 3390 for safety reasons. An increased initial window SHOULD NOT be turned on by default on systems without such monitoring capabilities.

The IETF TCPM working group is very much interested in further reports from experiments with this specification and encourages the publication of such measurement data. If no significant harm is reported, a follow-up document may revisit the question on whether a larger initial window can be safely used by default in all Internet hosts.

13. Related Proposals

Two other proposals [All10, Tou12] have been published to raise TCP’s initial window size over a large timescale. Both aim at reducing the uncertain impact of a larger initial window at an Internet wide
scale. Moreover, [Tou12] seeks an algorithm to automate the adjustment of IW safely over long haul period.

Although a modest, static increase of IW to 10 may address the near-term need for better web performance, much work is needed from the TCP research community to find a long term solution to the TCP flow startup problem.

14. Security Considerations

This document discusses the initial congestion window permitted for TCP connections. Although changing this value may cause more packet loss, it is highly unlikely to lead to a persistent state of network congestion or even a congestion collapse. Hence it does not raise any known new security issues with TCP.

15. 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 solution for the near-term deployment, while scaling IW over the long run remains a challenge for the TCP research community.

16. IANA Considerations

None

17. 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


Informative References


Gettys, J., "IW10 Considered Harmful", Internet-draft draft-gettys-iw10-considered-harmful-00, work in progress, August 2011.

[IW10] "TCP IW10 links", URL


    Simulation Study on Increasing TCP’s IW", Presented to 78th
    IRTF ICCRG working group meeting, July 2010. URL

    of IW and Initial RTO changes", Presented to 79th IETF TCPM
    working group meeting, Nov. 2010. URL

    Control Without a Startup Phase", Protocols for Fast, Long
    Distance Networks (PFLDnet) Workshop, February 2007. URL

    for speeding up web transfers", in Proceedings of IEEE

[PRAKS02] Partridge, C., Rockwell, D., Allman, M., Krishnan, R. and
    No. 8339, BBN Technologies, March 2002.

[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.


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

[RFC3150] Dawkins, S., Montenegro, G., Kojo, M. and V. Magret, "End-
    to-end Performance Implications of Slow Links", BCP 0048,

Chu, et. al.                  Expires August 2012               [Page 17]


Appendix A - List of Concerns and Corresponding Test Results

Concerns have been raised since this proposal was first published 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, additional tests have been conducted using either large scale clusters, simulations, or real testbeds. The following attempts to compile the list of concerns and summarize findings from relevant tests.

- How complete are various tests in covering many different traffic patterns?

The large scale Internet experiments conducted at Google front-end infrastructure covered a large portfolio of services beyond web search. It includes Gmail, Google Maps, Photos, News, Sites, Images, ..., etc, 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 results 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 tests conducted on this subject [Duk10, Sch11, Sch11-1, CW10] so far have shown only modest increase on packet drops. The only exception is from the testbed study [CW10] when under extremely high load and/or simultaneous opens. But under those conditions both IW=3 and IW=10 suffered very high packet loss rates though.

- 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 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 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. But a scheme to right size the initial window automatically over long timescales has yet to be developed.

o Need more thorough analysis of the impact on slow links

Although [Duk10] showed the large initial window reduced the average latency even for the dialup link class of only 56Kbps in bandwidth, more studies were needed in order to understand the effect of IW=10 on slow links at the microscopic level. [CW10] was conducted for this purpose.

Testbeds in [CW10] emulated a 300ms RTT, bottleneck link bandwidth as low as 64Kbps, and route queue size as low as 40 packets. A large combination of test parameters were used. Almost all tests showed varying degree of 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 the 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 the 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 the testbed experiments [CW10]. 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.

A 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.
Author’s Addresses

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

Acknowledgment

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

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) 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 small. Proportional Rate Reduction minimizes these excess window reductions such that at the end of recovery the actual window size will be as close as possible to ssthresh, 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.

Status of this Memo

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). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on August 27, 2012.
Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 3
2. Definitions .................................................. 4
3. Algorithms .................................................. 5
   3.1. Examples ................................................. 6
4. Properties .................................................... 9
5. Measurements ................................................ 11
6. Conclusion and Recommendations ........................... 12
7. Acknowledgements .......................................... 13
8. Security Considerations .................................... 13
9. IANA Considerations ....................................... 13
10. References .................................................. 13
Appendix A. Packet Conservation Bound ........................ 14
Authors’ Addresses ............................................. 15
1. Introduction

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) to improve the accuracy of the amount of data sent by TCP during loss recovery.

Standard Congestion Control [RFC5681] 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 wait for half window of ACKs to pass and then not receive any ACKs for the recovery and suffer a timeout.

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-halving is implemented in Linux after only being informally published [RHweb], including an uncompleted Internet-Draft [RHID]. Rate-halving does not adequately compensate for the implicit window reduction caused by the losses and assumes a 50% window reduction, which was completely standard at the time it was written, but not appropriate for modern congestion control algorithms such as Cubic [CUBIC], which can reduce the window by 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 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, ssthresh, 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 limits the total window reduction due to all mechanisms, including application stalls and the losses themselves.

We describe two slightly different reduction bound algorithms: conservative reduction bound (CRB), which is strictly packet conserving; and a slow start reduction bound (SSRB), which is more aggressive than CRB by at most one segment per ACK. PRR-CRB meets the strong conservative bound described in Appendix A, 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 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 taking more time to transmit them) it outperforms it, due to the lower probability of additional losses during recovery.

PRR and both reduction bounds 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 the same number of segments back 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 [RFC3517] and cwnd. This is what gives the algorithms their precision in the presence of events that cause uncertainty in other estimators.

We evaluated these and other algorithms in a large scale measurement study, summarized below. The most important results from that study are presented in an companion paper [IMC11]. PRR+SSRB outperforms both RFC 3517 and Linux Rate Halving under authentic network traffic, even though it is less aggressive than RFC 3517.

The algorithms are described as modifications to RFC 5681 [RFC5681], TCP Congestion Control, using concepts drawn from the pipe algorithm [RFC3517]. They are most accurate and more easily implemented with SACK [RFC2018], 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 and data rate.

We define some additional variables:

SACKd: The total number of bytes that the scoreboard indicates have 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. If SACK is not in use, SACKd is not defined.

DeliveredData: The total number of bytes that the current ACK indicates have been delivered to the receiver. When not in recovery, DeliveredData is the change in snd.una. With SACK, DeliveredData can be computed precisely as the change in snd.una plus the (signed) change in SACKd. 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

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) + MSS
    }
    // Attempt to catch up, as permitted by limit
    sndcnt = MIN(ssthresh − pipe, limit)
}

On any data transmission or retransmission:

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

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 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'retransmitted data would be sent. Note that the algorithms for deciding which data to send 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. Since PRR does not use cwnd during recovery it is not shown in the example.
RFC 3517

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 17 16 15 14 13 12 11 10 10 10 10 10 10 10 10 10
sent:   N  N  R

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 16 16 15 15 14 14 13 13 12 12 11 11 11
pipe:   19 19 18 18 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  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 11 10
sent:   N  N  R  N  N  N  N  N  N  N  N  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.
In this specific 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 the five ACKs trigger exactly one transmission each (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 all 15 lost segments. Rate Halving does not raise the window at all 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 in the event that 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 is 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, the excess window reduction causes it to take excessively long to recover the losses and exposes it 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, implementing 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 permitted by RFC 5681, which sends the same quantity of additional data as a single burst in response to the ACK that triggered Fast Retransmit.

For less extreme events, where 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 except as noted:

Proportional Rate Reduction maintains TCPs ACK clocking across most recovery events, including burst losses. RFC 3517 can send large unclocked bursts following burst losses.

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. Hypothetically, 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 bound 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 an 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 DeliveredData. Since short term errors in pipe are smoothed out across multiple ACKs and both Proportional Rate Reduction and the reduction bound converge to the same final window, errors in the pipe estimator have less impact on the final outcome.

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 with no cross traffic, the queue will maintain exactly constant length for the duration of the recovery, 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 companion IMC11 paper [IMC11] we describe some measurements comparing the various strategies for reducing the window during recovery. The results are summarized here.

The various window reduction algorithms and extensive instrumentation were all implemented in Linux 2.6. We used the uniform set of algorithms present in the base Linux implementation, including CUBIC [CUBIC], limited transmit [RFC3742], threshold transmit from [FACK] and lost retransmission detection algorithms. We confirmed that the behaviors of Rate Halving (the Linux default), RFC 3517 and PRR were authentic to their respective specifications and that 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 them.

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 30%, 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 it 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 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

Although the packet conserving bound is very appealing for a number of reasons, our measurements demonstrate that it is less aggressive and does not perform as well as RFC3517, which permits significant bursts of data when there are large bursts of losses. PRR-SSRB is a compromise that permits TCP to send one extra segment per ACK as relative 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. Even so, it often out performs RFC3517, because it avoids some of the self inflicted losses caused by bursts from RFC3517.

At this time we see no reason not to test and deploy PRR-SSRB on a large scale. Implementers worried about any potential impact of raising the window during recovery may want to optionally support PRR-CRB (which is actually simpler to implement) for comparison studies.

One final comment about terminology: we expect that common usage will drop "slow start reduction bound" from the algorithm name. This document needed to be pedantic about having distinct names for
proportional rate reduction and every variant of the reduction bound. However, once paired they become one.

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.

Ilpo Jarvinen for reviewing the code.

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 [Savage99], 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


Appendix A. Packet Conservation Bound

PRR-CRB meets a conservative, philosophically pure and aesthetically appealing notion of correct, described here. However, in real networks it does not perform as well as the algorithms described in RFC 3517, which proves to be non-conservative in a significant number of cases.

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 duration of the recovery, 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 if it is a full drop tail queue.

We demonstrate this property with a little thought experiment:...
Imagine a network path that has insignificant delays in both
directions, except for 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 bottleneck packet time:
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 DeliveredData 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. Losses or
reordering on the ACK path only cause wider fluctuations in the queue
size, but do not raise the peak size, independent of whether the data
is in order or out-of-order (including loss recovery from an earlier
RTT). Any more aggressive algorithm which sends additional data will
cause a queue overflow and loss. Any less aggressive algorithm will
under fill the queue. Therefore setting sndcnt to DeliveredData 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) may
increase the fine grained fluctuations in queue size but does not
change its basic behavior.

Note that the congestion control algorithm implements a broader
notion of optimal that includes appropriately sharing of the network.
Typical congestion control algorithms are likely to reduce the data
sent relative to the packet conserving bound implemented by PRR
bringing 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
This Internet-Draft, draft-ietf-tcpm-tcp-security-02.txt, has expired, and has been deleted from the Internet-Drafts directory. An Internet-Draft expires 185 days from the date that it is posted unless it is replaced by an updated version, or the Secretariat has been notified that the document is under official review by the IESG or has been passed to the RFC Editor for review and/or publication as an RFC. This Internet-Draft was not published as an RFC.

Internet-Drafts are not archival documents, and copies of Internet-Drafts that have been deleted from the directory are not available. The Secretariat does not have any information regarding the future plans of the author or working group, if applicable, with respect to this deleted Internet-Draft. For more information, or to request a copy of the document, please contact the author directly.

Draft Author:
Fernando Gont<fernando@gont.com.ar>
Abstract

The primary state variables used by all TCP congestion control algorithms, cwnd and ssthresh are heavily overloaded, carrying different semantics in different states. This leads to excess implementation complexity and poorly defined behaviors under some combinations of events, such as loss recovery during cwnd validation. We propose a new framework for TCP congestion control, and to recast current standard algorithms to use new state variables. This new framework will not generally change the behavior of any of the primary congestion control algorithms when invoked in isolation but will permit new algorithms with better behaviors in many corner cases, such as when two distinct primary algorithms are invoked concurrently. It will also foster the creation of new algorithms to address some events that are poorly treated by today’s standards. For the vast majority of traditional algorithms the transformation to the new state variables is completely straightforward. However, the resulting implementation will technically be in violation of all existing TCP standards, even if it is fully compliant with their principles and intent.

Status of this Memo

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). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on August 24, 2012.

Copyright Notice
Copyright (c) 2012 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
Provisions Relating to IETF Documents
(http://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 3
   1.1. Overview of the new algorithm ............................ 3
   1.2. Standards Impact ........................................... 4
   1.3. Meta Language .............................................. 5
2. State variables and definitions ................................. 5
3. Updated Algorithms ............................................. 6
   3.1. Congestion avoidance ....................................... 6
   3.2. Proportional Rate Reduction ............................... 7
   3.3. Restart after idle, Congestion Window Validation and
        Pacing ...................................................... 8
   3.4. RTO and F-RTO ............................................. 9
   3.5. Undo ...................................................... 9
   3.6. Control Block Interdependence ............................. 9
   3.7. New Reno ................................................ 9
4. Example Pseudocode ............................................. 10
5. Compatibility with existing implementations ................... 11
6. Security Considerations ....................................... 12
7. IANA Considerations .......................................... 13
8. References .................................................. 13
Author’s Address ................................................ 14
1. Introduction

The primary state variables used by all TCP congestion control algorithms, cwnd and ssthresh, are heavily overloaded, carrying different semantics in different states. This leads to excess implementation complexity and poorly defined behaviors under some combinations of events, such as overlapping application stalls and loss recovery. Multiple algorithms sharing the same state variables lead to excess complexity and conflicting correctness constraints, making it unreasonably difficult to implement, test and evaluate new algorithms.

We are proposing a new framework for TCP congestion control and it use new state variables that separate transmission scheduling, which determines precisely when data is sent, from congestion control, which determines the amount of data to be sent in each RTT. This separation greatly simplifies the interactions between the two subsystems and permits vast range of new algorithms that are not feasible with the current parameterization.

This note describes the new framework, represented through its state variables, and presents a preliminary mapping between current standards and new algorithms based on the new state variables. At this point the new algorithms are not fully specified, and many have still unconstrained design choices. In most cases, our goal is to precisely mimic today's standard TCP, at least as far as well defined primary behaviors. In general, it is a non-goal to mimic behaviors in poorly defined corner cases, or other cases where standard behaviors are viewed as being problematic.

It is called Laminar because one of its design goals is to eliminate unnecessary turbulence introduced by TCP itself.

1.1. Overview of the new algorithm

The new framework separate transmission scheduling, which determines precisely when data is sent, from Congestion Control, which determines the total amount of data sent in any given RTT.

The default algorithm for transmission scheduling is a strict implementation of Van Jacobson's packet conservation principle [Jacobson88]. Data arriving at the receiver cause ACKs which in turn cause the sender to transmit an equivalent quantity of data back into the network. The primary state variable is implicit in the quantity of data and ACKs circulating in the network. This state observed through a new "total_pipe" estimator, which is a generalization of "pipe" as described in RFC 3517. [RFC3517]
A new state variable, CCwin, is the primary congestion control state variable. It is updated only by the congestion control algorithms, which are concerned with detecting and regulating the overall level of congestion along the path. CCwin is TCP’s best estimate for an appropriate average window size. In general, it rises when the network seem to be underfilled and is reduced in the presence of congestion signals, such as loss, ECN marks or increased delay. Although CCwin resembles cwnd, it is actually quite different, for one thing the new parameterization does not use ssthresh at all.

Any time CCwin is larger than total_pipe, the default algorithm to grow total_pipe is for each ACK to trigger one segment of additional data. This is essentially an implicit slowstart, but it is gated by the difference between CCwin and total_pipe, rather than the difference between cwnd and ssthresh.

During Fast Retransmit, the congestion control algorithm, such as CUBIC, generally reduces CCwin in a single step. Proportional Rate Reduction [PRR] is used to gradually reduce total_pipe to agree with CCwin. PRR is based on Laminar principles, so its specification has many parallels to this document.

Connection startup is accomplished as follows: CCwin is set to MAX_WINDOW (akin to ssthresh), and IW segments are transmitted. The ACKs from these segments trigger additional data transmissions, and slowstart proceeds as it does today. The very first congestion event is a special case because there is not a prior value for CCwin. By default on the first congestion event only, CCwin would be set from total_pipe, and then standard congestion control is invoked.

The primary advantage of the Laminar framework is that by partitioning congestion control and transmission scheduling into separate subsystems, each is subject to far simpler design constraints, making it far easier to develop many new algorithms that are not feasible with the current organization of the code.

1.2. Standards Impact

Since we are proposing to refactor existing standards into new state variables, all of the current congestion control standards documents will potentially need to be revised. Note that there are roughly 60 RFC that mention cwnd or ssthresh, and all of them should be reviewed for material that may need to be updated.

This document does not propose to change the TCP friendly paradigm. By default all updated algorithms using these new state variables would have behaviors similar to the current TCP implementations. We do however anticipate some second order effects which we will address...
in section XXX below. For example while testing PRR it was observed that suppressing bursts by slightly delaying transmissions can improve average performance, even though in a strict sense the new algorithm is less aggressive than the old.

1.3. Meta Language

We use the following terms when describing algorithms and their alternatives:

Standard - The current state of the art, including both formal standards and widely deployed algorithms that have come into standard use, even though they may not be formally specified. [Although PRR does not yet technically meet these criteria, we include it here].

default - The simplest or most straightforward algorithm that fits within the Laminar framework. For example implicit slowstart whenever total_pipe is less than CCwin. This term does not make a statement about the relative aggressiveness or any other properties of the algorithm except that it is a reasonable choice and straightforward to implement.

conformant - An algorithm that can produce the same packet trace as a TCP implementation that strictly conforms to the current standards.

mimic - An algorithm constructed to be conformant to standards.

opportunity - An algorithm that can do something better than the standard algorithm, typically better behavior in a corner cases that is either not well specified or where the standard behavior is viewed as being less than ideal.

more/less aggressive - Any algorithm that sends segments earlier/later than another (typically conformant) algorithm under identical sequences of events. Note that this is an evaluation of the packet level behavior, and does not reflect any higher order effects.

Net more/less aggressive - Any algorithm that gets more/less average data rate than another (typically conformant) algorithm. This is an empirical statement based on measurement (or perhaps justified speculation), and potentially indicates a problem with failing to be "TCP friendly".

2. State variables and definitions

CCwin - The primary congestion control state variable.
DeliveredData - The total number of bytes that the current ACK indicates have been delivered to the receiver. (See PRR for more detail).

total_pipe - The total quantity of circulating data and ACKs. In addition to RFC 3517 pipe, it includes DeliveredData for the current ack, plus any data held for delayed transmission, for example to permit a later TSO transmission.

sendcnt - The quantity of data to be sent in response to the current event.

application stall - The application is failing to keep TCP in bulk mode: either the sender is running out of data to send, or the receiver is not reading it fast enough. When there is an application stall, congestion control does not regulate data transmission and some of the protocol events are triggered by application reads or writes, as appropriate.

3. Updated Algorithms

A survey of standard, common and proposed algorithms, and how they might be reimplemented under the Laminar framework.

3.1. Congestion avoidance

Under the Laminar framework the loss recovery mechanism does not, by default, interfere with the primary congestion control algorithms. The CCwin state variable is updated only by the algorithms that decide how much data to send on successive round trips. For example standard Reno AIMD congestion control [RFC5681] can be implemented by raising CCwin by one segment every CCwin worth of ACKs (once per RTT) and halving it on every loss or ECN signal (e.g. CCwin = CCwin/2). During recovery the transmission scheduling part of the Laminar framework makes the necessary adjustments to bring total_pipe to agree with CCwin, without tampering with CCwin.

This separation between computing CCwin and transmission scheduling will enable new classes of congestion control algorithms, such as fluid models that adjust CCwin on every ACK, even during recovery. This is safe because raising CCwin does not directly trigger any transmissions, it just steers the transmission scheduling closer to the end of recovery. Fluid models have a number of advantages, such as simpler closed form mathematical representations, and are intrinsically more tolerant to reordering since non-recovery disordered states don’t inhibit growing the window.
Investigating alternative algorithms and their impact is out of scope for this document. It is important to note that while our goal here is not to alter the TCP friendly paradigm, Laminar does not include any implicit or explicit mechanism to prevent a Tragedy of the Commons. However, see the comments in Section 6.

The initial slowstart does not use the CCwin, except that CCwin starts at the largest possible value. It is the transmission scheduling algorithms that are responsible for performing the slowstart. On the first loss it is necessary to compute a reasonable CCwin from total_pipe. Ideally, we might save total_pipe at the time each segment is scheduled for transmission, and use the saved value associated with the lost segment to prime CCwin. However, this approach requires extra state attached to every segment in the retransmit queue. A simpler approach is to have a mathematical model the slowstart, and to prime CCwin from total_pipe at the time the loss is detected, but scaled down by the effective slowstart multiplier (e.g. 1.5 or 2). In either case, once CCwin is primed from total_pipe, it is typically appropriate to invoke the reduction on loss function, to reduce it again per the congestion control algorithm.

Nearly all congestion control algorithms need to have some mechanism to prevent CCwin from growing while it is not regulating transmissions e.g. during application stalls.

3.2. Proportional Rate Reduction

Since PRR [I-D.ietf-tcpm-proportional-rate-reduction] was designed with Laminar principles in mind, updating it is a straightforward variable substitution. CCwin replaces ssthresh, and RecoverFS is initialized from total_pipe at the beginning of recovery. Thus PRR provides a gradual window reduction from the prior total_pipe down to the new CCwin.

There is one important difference from the current standards: CCwin is computed solely on the basis of the prior value of CCwin. Compare this to RFC 5681 which specifies that the congestion control function is computed on the basis of the FlightSize (e.g. ssthresh=FlightSize/2). This change from prior standard completely alters how application stalls interact with congestion control.

Consider what happens if there is an application stall for most of the RTT just before a Fast Retransmit: Under Laminar it is likely that CCwin will be set to a value that is larger than total_pipe, and subject to available application data PRR will go directly to slowstart mode, to raise total_pipe up to CCwin. Note that the final CCwin value does not depend on the duration of the application stall.
With standard TCP, any application stall reduces the final value of cwnd at the end of recovery. In some sense application stalls during recovery are treated as though they are additional losses, and have a detrimental effect on the connection data rate that lasts far longer than the stall itself.

If there are no application stalls, the standard and Laminar variants of the PRR algorithm should have identical behaviors. Although it is tempting to characterize Laminar as being more aggressive than the standards, it would be more apropos to characterize the standard as being excessively timid under common combinations of overlapping events that are not well represented by benchmarks or models.

3.3. Restart after idle, Congestion Window Validation and Pacing

Decoupling congestion control from transmission scheduling permits us to develop new algorithms to raise total_pipe to CCwin after an application stall or other events. Although it was stated earlier that the default transmission scheduling algorithm for raising total_pipe is an implicit slowstart, there is lots of opportunity for better algorithms.

We imagine a new class of hybrid transmission scheduling algorithms that use a combination of pacing and slowstart to reestablish TCP’s self clock. For example, whenever total_pipe is significantly below CCwin, RTT and CCwin can be used to directly compute a pacing rate. We suspect that pacing at the previous full rate will prove to be somewhat brittle, yielding erratic results. It is more likely that a hybrid strategy will work better, for example by pacing at some fraction (1/2 or 1/4) of the prior rate until total_pipe reaches some fraction of CCwin (e.g. CCwin/2) and then using conventional slowstart to bring total_pipe the rest of the way up to CCwin.

This is far less aggressive than standard TCP without cwnd validation [RFC2861] or when the application stall was less than one RTO, since standards permit TCP to send a full cwnd size burst in these situations. It is potentially more aggressive than conventional slowstart invoked by cwnd validation when the application stall is longer than several RTOs. Both standard behaviors in these situations have always been viewed as problematic, because interface rate bursts are clearly too aggressive and a full slowstart is clearly too conservative. Mimicking either is a non-goal, when there is ample opportunity to find a better compromise.

Although strictly speaking any new transmission scheduling algorithms are independent of the Laminar framework, they are expected to have substantially better behavior in many common environments and as such strongly motivate the effort required to refactor TCP implementations.
and standards.

3.4. RTO and F-RTO

We are not proposing any changes to the RTO timer or the F-RTO [RFC5682] algorithm used to detect spurious retransmissions. Once it is determined that segments were lost, CCwin is updated to a new value as determined by the congestion control function, and Laminar implicit slowstart is used to clock out (re)transmissions. Once all holes are filled, a hybrid paced transmissions can be used to reestablish TCPs self clock at the new data rate. This can be the same hybrid pacing algorithm as is used to recover the self clock after application stalls.

Note that as long as there is non-contiguous data at the receiver the retransmission algorithms require timely SACK information to make proper decisions about which segments to send. Pacing during loss recovery is not recommended without further investigation.

3.5. Undo

Since CCwin is not used to implement transmission scheduling, undo is trivial. CCwin can just be set back to a prior value and the transmission scheduling algorithm will transmit more (or less) data as needed.

3.6. Control Block Interdependence

Under the Laminar framework, congestion control state can be easily shared between connections [RFC2140]. An ensemble of connections can each maintain their own total_pipe (partial_pipe?) which in aggregate tracks a single common CCwin. A master transmission scheduler allocates permission to send (sndcnt) to each of the constituent connection on the basis of the difference between the CCwin and the aggregate total_pipe, and a fairness or capacity allocation policy that balances the flows. Note that ACKs on one connection in an ensemble might be used to clock transmissions on another connection, and that following a loss, the window reductions can be allocated to flows other than the one experiencing the loss.

3.7. New Reno

The key to making Laminar function well without SACK is having good estimators for DeliveredData and total_pipe. By definition every duplicate ACK indicates that one segment has arrived at the receiver and total_pipe has fallen by one. On any ACK that advances snd.una, total_pipe can be updated from snd.nxt-snd.una, and DeliveredData is the change in snd.una, minus the estimated DeliveredData of the
preceding duplicate ACKs.

4. Example Pseudocode

The example pseudocode in this section incorporates (or subsumes) the following algorithms:

On startup:

\[
\begin{align*}
&\text{CCwin} = \text{MAX_WINOW} \\
&\text{sndBank} = \text{IW}
\end{align*}
\]

On every ACK:

\[
\begin{align*}
&\text{DeliveredData} = \text{delta(snd.una)} + \text{delta(SACKd)} \\
&\text{pipe} = \text{(RFC 3517 pipe algorithm)} \\
&\text{total_pipe} = \text{pipe+DeliveredData+sndBank} \\
&\text{sndcnt} = \text{DeliveredData} \quad \text{// Default outcome}
\end{align*}
\]

\[
\begin{align*}
\text{if new_recovery()}: \\
&\quad \text{if CCwin == MAX_WIN:} \\
&\quad &\quad \text{CCwin} = \frac{\text{total_pipe}}{2} \quad \text{// First time only} \\
&\quad &\quad \text{CCwin} = \frac{\text{CCwin}}{2} \quad \text{// Reno congestion control} \\
&\quad &\quad \text{prr_delivered} = 0 \quad \text{// Total bytes delivered during recov} \\
&\quad &\quad \text{prr_out} = 0 \quad \text{// Total bytes sent during recovery} \\
&\quad &\quad \text{RecoverFS} = \text{total_pipe} \quad \text{//}
\end{align*}
\]

\[
\begin{align*}
\text{if !in_recov()} \&\& \text{!application_limited()}: \\
&\quad \text{CCwin} += \frac{\text{MSS}}{\text{CCwin}} \\
&\quad \text{prr_delivered} += \text{DeliveredData} \quad \text{// noop if not in recovery}
\end{align*}
\]
if total_pipe > CCwin:
    // Proportional Rate Reduction
    sndcnt = CEIL(prr_delivered * CCwin / RecoverFS) - prr_out

else if total_pipe < CCwin:
    if in_recovery():
        // PRR Slow Start Reduction Bound
        limit = MAX(prr_delivered - prr_out, DeliveredData) + SMSS
        sndcnt = MIN(CCwin - total_pipe, limit)
    else:
        // slow start with appropriate byte counting
        inc = MIN(DeliveredData, 2*MSS)
        sndcnt = DeliveredData + inc

    // cue the (re)transmission machinery
    sndBank += sndcnt
    limit = maxBank()
    if sndBank > limit:
        sndBank = limit
    tcp_output()

For any data transmission or retransmission:

tcp_output():
    while sndBank && tso_ok():
        len = sendsomething()
        sndBank -= len
        prr_out += len // noop if not in recovery

5. Compatibility with existing implementations

On a segment by segment basis, the above algorithm is [believed to be] fully conformant with or less aggressive than standards under all conditions.

However this condition is not sufficient to guarantee that average performance can’t be substantially better (net more aggressive) than standards. Consider an application that keeps TCP in bulk mode nearly all of the time, but has occasional pauses that last some fraction of one RTT. A fully conformant TCP would be permitted to "catch up" by sending a partial window burst at full interface rate. In some networks, such bursts might be very disruptive, causing otherwise unnecessary packet losses and corresponding cwnd reductions.
In Laminar, such a burst would be permitted, but the default algorithm would be slowstart. A better algorithm would be to pace the data at (some fraction of) the prior rate. Neither pacing nor slowstart is likely to cause unnecessary losses, and as was observed while testing PRR, being less aggressive at the segment level has the potential to increase average performance [IMC11PRR]. In this scenario Laminar with pacing has the potential to outperform both of the behaviors described by standards.

6. Security Considerations

The Laminar framework does not change the risk profile for TCP (or other transport protocols) themselves.

However, the complexity of current algorithms as embodied in today’s code present a substantial barrier to people wishing to cheat "TCP friendliness". It is a fairly well known and easily rediscovered result that custom tweaks to make TCP more aggressive in one environment generally make it fragile and perform less well across the extreme diversity of the Internet. This negative outcome is a substantial intrinsic barrier to wide deployment of rogue congestion control algorithms.

A direct consequence of the changes proposed in this note, decoupling congestion control from other algorithms, is likely to reduce the barrier to rogue algorithms. However this separation and the ability to introduce new congestion control algorithms is a key part of the motivation for this work.

It is also important to note that web browsers have already largely defeated TCP’s ability to regulate congestion by opening many concurrent connections. When a Web page contains content served from multiple domains (the norm these days) all modern browsers open between 35 and 60 connections (see: http://www.browserscope.org/?category=network ). This is the Web community’s deliberate workaround for TCP’s perceived poor performance and inability fill certain kinds of consumer grade networks. As a consequence the transport layer has already lost a substantial portion of its ability to regulate congestion. It was not anticipated that the tragedy of the commons in Internet congestion would be driven by competition between applications and not TCP implementations.

In the short term, we can continue to try to use standards and peer pressure to moderate the rise in overall congestion levels, however the only real solution is to develop mechanisms in the Internet itself to apply some sort of backpressure to overly aggressive
applications and transport protocols. We need to redouble efforts by the ConEx WG and others to develop mechanisms to inform policy with information about congestion and its causes. Otherwise we have a looming tragedy of the commons, in which TCP has only a minor role.

Implementers that change Laminar from counting bytes to segments have to be cautious about the effects of ACK splitting attacks [Savage99], where the receiver acknowledges partial segments for the purpose of confusing the sender’s congestion accounting.

7. IANA Considerations

This document makes no request of IANA.

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

8. References


Author’s Address
Matt Mathis
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA
Email: mattmathis@google.com
Abstract

When NAT is used in address resource restricted environment, or when a lot of users are located under a NAT device, IP address and port resources may be eaten up, and it brings severe bad effects on user experiences. This situation can be greatly mitigated by tweaking mapping behavior and session timer handling at NAT function. This document proposes to NAT IP address and port resource optimizing extension for address resource restricted environment. One extension is to enable simultaneous use of a port for different transport sessions, and the other is to make use of TCP timestamp for TIME_WAIT Assassination.

Status of this Memo

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). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on August 4, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents.
1. Introduction

The internet has run out of IPv4 addresses, and after that IPv4 addresses that can be allocated to a user will be much more restricted. NAT is a tool that is widely used for this problem of IPv4 address shortage. However, there will be more and more demands for resources to provide the Internet access to the users and devices. IPv6 is a fundamental solution for this problem, but the deployment of IPv6 takes time.

In some cases, e.g. browsing a dynamic web page for a map service, a lot of sessions are used by the browser, and a number of ports are eaten up in short time. What is worse is that when a NAT is in between, the NAT keeps track of each session for long time, typically for four minutes, even if the session is terminated by both ends within a few seconds.

This problem is caused or worsened by the following behaviors of NAT.

1. In a lot of NAT implementation, a port that is available at NAT is allocated or a transport session. i.e. a NAT does not use a port for multiple sessions simultaneously.

2. TCP TIME_WAIT state requires 2*MSL wait before a session is closed, so at a NAT device many session states are kept, even if both ends of a session have closed and deleted the state for the session.

We propose two mechanisms to change the above behaviors of NAT that enables to save addresses and ports resources.
1.1. TCP TIME_WAIT

TCP TIME_WAIT mechanism is written in RFC793. TCP TIME_WAIT status requires 2*MSL to wait before connection to be CLOSED, in case:

1. Packet in previous session be transfered later
2. Retransmission of ACK packet (which is respond to FIN) is needed.

In case 1, the old packet should be discarded, not to harm new session.
In case 2, retransmission should be done to close destination port.
If TIME_WAIT mechanism do not work, TCP transmission problem may happen.

1.2. TIME_WAIT Assassination

The server may accept the TCP SYN from the client change the state of the port from TCP TIME-WAIT to TCP Established. This is known as TIME-WAIT assassination.

1.3. PAWS (Protect Against Wrapped Sequence numbers)

PAWS (Protect Against Wrapped Sequence numbers) is written in RFC1323. PAWS is used to prevent packet in previous session be transfered to the new session.

2. Proposal

2.1. Adopt Intended Port over-use mechanism

To use single port for several connections, over-use port. For example, if destination address differ in two connections, use single port for NAT port assignment. In such case, this mechanism SHOULD meet the requirement written in RFC5382. i.e. REQ-1: A NAT MUST have an "Endpoint-Independent Mapping" behavior for TCP.
(There may be some other violations to adopt this, which I’ll consider and write by IETF83 Paris)

2.2. Adopt RFC6191 to NAT

To make connection re-usable for TCP transmission (e.g. http transmission) in address resource restricted network, force port state to be CLOSED, as soon as a session have finished (i.e. port is in TCP TIME_WAIT state), if there are any SYN-WAIT state for new session, waiting for port to be released.
This mechanism is written in RFC6191.
To make this mechanism work in network that several PCs connected to the NAT equipment, value of timestamp SHOULD be successive.
(timestamps MAY be overwritten in NAT equipment, so that timestamp in each packet that go through NAT equipment be successive.)
Also, consideration of some transmission pattern and effect is needed.
(I’ll consider patterns, and write by IETF83 Paris)

3. Security Considerations
(I’ll write some by IETF83, Paris.)

4. Normative References


Authors’ Addresses

Kengo Naito
NTT SI Lab
3-9-11 Midori-Cho
Musashino-shi, Tokyo  180-8585
Japan
Phone: +81 422 59 4949
Email: naito.kengo@lab.ntt.co.jp

Arifumi Matsumoto
NTT SI Lab
3-9-11 Midori-Cho
Musashino-shi, Tokyo  180-8585
Japan
Phone: +81 422 59 3334
Email: arifumi@nttv6.net