TCP Maintenance (TCPM)
Internet-Draft
Intended status: Standards Track
Expires: October 14, 2013

D. Borman
Quantum Corporation
B. Braden
University of Southern California
V. Jacobson
Packet Design
R. Scheffenegger, Ed.
NetApp, Inc.
April 12, 2013

TCP Extensions for High Performance
draft-ietf-tcpm-1323bis-09

Abstract

This document specifies a set of TCP extensions to improve performance over paths with a large bandwidth * delay product and to provide reliable operation over very high-speed paths. It defines TCP options for scaled windows and timestamps. The timestamps are used for two distinct mechanisms, RTTM (Round Trip Time Measurement) and PAWS (Protection Against Wrapped Sequences).

This document updates and obsoletes RFC 1323.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The TCP protocol [RFC0793] was designed to operate reliably over almost any transmission medium regardless of transmission rate, delay, corruption, duplication, or reordering of segments. Over the years, advances in networking technology has resulted in ever-higher transmission speeds, and the fastest paths are well beyond the domain for which TCP was originally engineered.

This document defines a set of modest extensions to TCP to extend the domain of its application to match the increasing network capability. It is an update to and obsoletes [RFC1323], which in turn is based upon and obsoletes [RFC1072] and [RFC1185].

Changes between [RFC1323] and this document are detailed in Appendix G.

For brevity, the full discussions of the merits and history behind the TCP options defined within this document have been omitted. [RFC1323] should be consulted for reference. It is recommended that a modern TCP stack implements and make use of the extensions described in this document.

1.1. TCP Performance

TCP performance problems arise when the bandwidth * delay product is large. A network having such paths is referred to as "long, fat network" (LFN).

There are three fundamental performance problems with basic TCP over LFN paths:

(1) Window Size Limit

The TCP header uses a 16 bit field to report the receive window size to the sender. Therefore, the largest window that can be used is 2^16 = 65K bytes.

To circumvent this problem, Section 2 of this memo defines a TCP option, "Window Scale", to allow windows larger than 2^16. This option defines an implicit scale factor, which is used to multiply the window size value found in a TCP header to obtain the true window size.

(2) Recovery from Losses

Packet losses in an LFN can have a catastrophic effect on throughput.

To generalize the Fast Retransmit/Fast Recovery mechanism to handle multiple packets dropped per window, selective acknowledgments are required. Unlike the normal cumulative acknowledgments of TCP, selective acknowledgments give the sender a complete picture of which segments are queued at the receiver and which have not yet arrived.

Selective acknowledgements are specified in a separate document, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP" [RFC6675], and not further discussed in this document.

(3) Round-Trip Measurement

TCP implements reliable data delivery by retransmitting segments that are not acknowledged within some retransmission timeout (RTO) interval. Accurate dynamic determination of an appropriate RTO is essential to TCP performance. RTO is determined by estimating the mean and variance of the measured round-trip time (RTT), i.e., the time interval between sending a segment and receiving an acknowledgment for it [Jacobson88a].

Section 3.2 defines a TCP option, "Timestamp", and then specifies a mechanism using this option that allows nearly every segment, including retransmissions, to be timed at negligible computational cost. We use the mnemonic RTTM (Round Trip Time Measurement) for this mechanism, to distinguish it from other uses of the Timestamp Option.

1.2. TCP Reliability

An especially serious kind of error may result from an accidental reuse of TCP sequence numbers in data segments. TCP reliability depends upon the existence of a bound on the lifetime of a segment: the "Maximum Segment Lifetime" or MSL.

Duplication of sequence numbers might happen in either of two ways:

(1) Sequence number wrap-around on the current connection

A TCP sequence number contains 32 bits. At a high enough transfer rate, the 32-bit sequence space may be "wrapped" (cycled) within the time that a segment is delayed in queues.

(2) Earlier incarnation of the connection

Suppose that a connection terminates, either by a proper close sequence or due to a host crash, and the same connection (i.e.,

using the same pair of port numbers) is immediately reopened. A delayed segment from the terminated connection could fall within the current window for the new incarnation and be accepted as valid.

Duplicates from earlier incarnations, case (2), are avoided by enforcing the current fixed MSL of the TCP specification, as explained in Section 4.8 and Appendix B. However, case (1), avoiding the reuse of sequence numbers within the same connection, requires an upper bound on MSL that depends upon the transfer rate, and at high enough rates, a dedicated mechanism is required.

A possible fix for the problem of cycling the sequence space would be to increase the size of the TCP sequence number field. For example, the sequence number field (and also the acknowledgment field) could be expanded to 64 bits. This could be done either by changing the TCP header or by means of an additional option.

Section 4 presents a different mechanism, which we call PAWS (Protection Against Wrapped Sequence numbers), to extend TCP reliability to transfer rates well beyond the foreseeable upper limit of network bandwidths. PAWS uses the TCP timestamp option defined in Section 3.2 to protect against old duplicates from the same connection.

1.3. **Using TCP options**

The extensions defined in this document all use TCP options.

When [RFC1323] was published, there was concern that some buggy TCP implementation might be crashed by the first appearance of an option on a non-<SYN> segment. However, bugs like that can lead to DOS attacks against a TCP, so it is now expected that most TCP implementations will properly handle unknown options on non-<SYN> segments. But it is still prudent to be conservative in what you send, and avoiding buggy TCP implementation is not the only reason for negotiating TCP options on <SYN> segments.

The window scale option negotiates fundamental parameters of the TCP session. Therefore, it is only sent during the initial handshake. Furthermore, the window scale option will be sent in a <SYN,ACK> segment only if the corresponding option was received in the initial <SYN> segment.

The timestamp option may appear in any data or <ACK> segment, adding 12 bytes to the 20-byte TCP header. We recognize there is a trade-off between the bandwidth saved by reducing unnecessary retransmission timeouts, and the extra header bandwidth used by this

option. It is required that this TCP option will be sent on non-<SYN> segments only after an exchange of options on the <SYN> segments has indicated that both sides understand this extension.

Appendix A contains a recommended layout of the options in TCP headers to achieve reasonable data field alignment.

Finally, we observe that most of the mechanisms defined in this memo are important for LFN's and/or very high-speed networks. For low-speed networks, it might be a performance optimization to NOT use these mechanisms. A TCP vendor concerned about optimal performance over low-speed paths might consider turning these extensions off for low-speed paths, or allow a user or installation manager to disable them.

1.4. 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 [RFC2119].

In this document, these words will appear with that interpretation only when in UPPER CASE. Lower case uses of these words are not to be interpreted as carrying [RFC2119] significance.

2. TCP Window Scale Option

2.1. Introduction

The window scale extension expands the definition of the TCP window to 32 bits and then uses a scale factor to carry this 32-bit value in the 16-bit Window field of the TCP header (SEG.WND in RFC 793). The scale factor is carried in a TCP option, Window Scale. This option is sent only in a <SYN> segment (a segment with the SYN bit on), hence the window scale is fixed in each direction when a connection is opened.

The maximum receive window, and therefore the scale factor, is determined by the maximum receive buffer space. In a typical modern implementation, this maximum buffer space is set by default but can be overridden by a user program before a TCP connection is opened. This determines the scale factor, and therefore no new user interface is needed for window scaling.

2.2. Window Scale Option

The three-byte Window Scale option MAY be sent in a <SYN> segment by a TCP. It has two purposes: (1) indicate that the TCP is prepared to do both send and receive window scaling, and (2) communicate a scale factor to be applied to its receive window. Thus, a TCP that is prepared to scale windows SHOULD send the option, even if its own scale factor is 1. The scale factor is limited to a power of two and encoded logarithmically, so it may be implemented by binary shift operations.

TCP Window Scale Option (WSopt):

Kind: 3

Length: 3 bytes

```
+---------+---------+---------+
| Kind=3  |Length=3 |shift.cnt|
+---------+---------+---------+
     1         1         1
```

This option is an offer, not a promise; both sides MUST send Window Scale options in their <SYN> segments to enable window scaling in either direction. If window scaling is enabled, then the TCP that sent this option will right-shift its true receive-window values by 'shift.cnt' bits for transmission in SEG.WND. The value 'shift.cnt' MAY be zero (offering to scale, while applying a scale factor of 1 to the receive window).

This option MAY be sent in an initial <SYN> segment (i.e., a segment with the SYN bit on and the ACK bit off). It MAY also be sent in a <SYN,ACK> segment, but only if a Window Scale option was received in the initial <SYN> segment. A Window Scale option in a segment without a SYN bit SHOULD be ignored.

The window field in a segment where the SYN bit is set (i.e., a <SYN> or <SYN,ACK>) is never scaled.

2.3. Using the Window Scale Option

A model implementation of window scaling is as follows, using the notation of [RFC0793]:

o All windows are treated as 32-bit quantities for storage in the connection control block and for local calculations. This includes the send-window (SND.WND) and the receive-window (RCV.WND) values, as well as the congestion window.

o The connection state is augmented by two window shift counts, Snd.Wind.Scale and Rcv.Wind.Scale, to be applied to the incoming and outgoing window fields, respectively.

o If a TCP receives a <SYN> segment containing a Window Scale option, it sends its own Window Scale option in the <SYN,ACK> segment.

o The Window Scale option is sent with shift.cnt = R, where R is the value that the TCP would like to use for its receive window.

o Upon receiving a <SYN> segment with a Window Scale option containing shift.cnt = S, a TCP sets Snd.Wind.Scale to S and sets Rcv.Wind.Scale to R; otherwise, it sets both Snd.Wind.Scale and Rcv.Wind.Scale to zero.

o The window field (SEG.WND) in the header of every incoming segment, with the exception of <SYN> segments, is left-shifted by Snd.Wind.Scale bits before updating SND.WND:

```
SND.WND = SEG.WND << Snd.Wind.Scale
```

(assuming the other conditions of [RFC0793] are met, and using the "C" notation "<<" for left-shift).

o The window field (SEG.WND) of every outgoing segment, with the exception of <SYN> segments, is right-shifted by Rcv.Wind.Scale bits:

SND.WND = RCV.WND >> Rcv.Wind.Scale

TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^31 bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice versa, the left edge of the sender's window has to be at most 2^31 away from the right edge of the receiver's window. Similarly with the sender's right edge and receiver's left edge. Since the right and left edges of either the sender's or receiver's window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that two times the max window size must be less than 2^31, or

max window < 2^30

Since the max window is 2^S (where S is the scaling shift count) times at most 2^16 - 1 (the maximum unscaled window), the maximum window is guaranteed to be < 2^30 if S <= 14. Thus, the shift count MUST be limited to 14 (which allows windows of 2^30 = 1 Gbyte). If a Window Scale option is received with a shift.cnt value exceeding 14, the TCP SHOULD log the error but use 14 instead of the specified value.

The scale factor applies only to the Window field as transmitted in the TCP header; each TCP using extended windows will maintain the window values locally as 32-bit numbers. For example, the "congestion window" computed by Slow Start and Congestion Avoidance is not affected by the scale factor, so window scaling will not introduce quantization into the congestion window.

2.4. Addressing Window Retraction

When a non-zero scale factor is in use, there are instances when a retracted window can be offered - see Appendix F for a detailed example. The end of the window will be on a boundary based on the granularity of the scale factor being used. If the sequence number is then updated by a number of bytes smaller than that granularity, the TCP will have to either advertise a new window that is beyond what it previously advertised (and perhaps beyond the buffer), or will have to advertise a smaller window, which will cause the TCP window to shrink. Implementations MUST ensure that they handle a shrinking window, as specified in section 4.2.2.16 of [RFC1122].

For the receiver, this implies that:

1) The receiver MUST honor, as in-window, any segment that would have been in-window for any <ACK> sent by the receiver.

2) When window scaling is in effect, the receiver SHOULD track the actual maximum window sequence number (which is likely to be greater than the window announced by the most recent <ACK>, if more than one segment has arrived since the application consumed any data in the receive buffer).

On the sender side:

3) The initial transmission MUST be within the window announced by the most recent <ACK>.

4) On first retransmission, or if the sequence number is out-of-window by less than (2^Rcv.Wind.Scale) then do normal retransmission(s) without regard to receiver window as long as the original segment was in window when it was sent.

5) Subsequent retransmissions MAY only be sent, if they are within the window announced by the most recent <ACK>.

3. RTTM -- Round-Trip Time Measurement

3.1. Introduction

Accurate and current RTT estimates are necessary to adapt to changing traffic conditions and to avoid an instability known as "congestion collapse" [RFC0896] in a busy network. However, accurate measurement of RTT may be difficult both in theory and in implementation.

Many TCP implementations base their RTT measurements upon a sample of one segment per window or less. While this yields an adequate approximation to the RTT for small windows, it results in an unacceptably poor RTT estimate for a LFN. If we look at RTT estimation as a signal processing problem (which it is), a data signal at some frequency, the packet rate, is being sampled at a lower frequency, the window rate. This lower sampling frequency violates Nyquist's criteria and may therefore introduce "aliasing" artifacts into the estimated RTT [Hamming77].

A good RTT estimator with a conservative retransmission timeout calculation can tolerate aliasing when the sampling frequency is "close" to the data frequency. For example, with a window of 8 segments, the sample rate is 1/8 the data frequency -- less than an order of magnitude different. However, when the window is tens or hundreds of segments, the RTT estimator may be seriously in error, resulting in spurious retransmissions.

If there are dropped segments, the problem becomes worse. Zhang [Zhang86], Jain [Jain86] and Karn [Karn87] have shown that it is not possible to accumulate reliable RTT estimates if retransmitted segments are included in the estimate. Since a full window of data will have been transmitted prior to a retransmission, all of the segments in that window will have to be ACKed before the next RTT sample can be taken. This means at least an additional window's worth of time between RTT measurements and, as the error rate approaches one per window of data (e.g., 10^-6 errors per bit for the Wideband satellite network), it becomes effectively impossible to obtain a valid RTT measurement.

A solution to these problems, which actually simplifies the sender substantially, is as follows: using TCP options, the sender places a timestamp in each data segment, and the receiver reflects these timestamps back in <ACK> segments. Then a single subtract gives the sender an accurate RTT measurement for every <ACK> segment (which will correspond to every other data segment, with a sensible receiver). We call this the RTTM (Round-Trip Time Measurement) mechanism.

It is vitally important to use the RTTM mechanism with big windows; otherwise, the door is opened to some dangerous instabilities due to aliasing. Furthermore, the option is probably useful for all TCP's, since it simplifies the sender.

3.2. TCP Timestamp Option

TCP is a symmetric protocol, allowing data to be sent at any time in either direction, and therefore timestamp echoing may occur in either direction. For simplicity and symmetry, we specify that timestamps always be sent and echoed in both directions. For efficiency, we combine the timestamp and timestamp reply fields into a single TCP Timestamp Option.

TCP Timestamp Option (TSopt):

Kind: 8

Length: 10 bytes

```
       +-------+-------+---------------------+---------------------+
       |Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
       +-------+-------+---------------------+---------------------+
           1       1              4                     4
```

The Timestamp Option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option.

The Timestamp Echo Reply field (TSecr) is valid if the ACK bit is set in the TCP header; if it is valid, it echoes a timestamp value that was sent by the remote TCP in the TSval field of a Timestamp option. When TSecr is not valid, its value MUST be zero. However, a value of zero does not imply TSecr being invalid. The TSecr value will generally be from the most recent Timestamp Option that was received; however, there are exceptions that are explained below.

A TCP MAY send the Timestamp option (TSopt) in an initial <SYN> segment (i.e., segment containing a SYN bit and no ACK bit), and MAY send a TSopt in other segments only if it received a TSopt in the initial <SYN> or <SYN,ACK> segment for the connection.

Once TSopt has been successfully negotiated (sent and received) during the <SYN>, <SYN,ACK> exchange, TSopt MUST be sent in every non-<RST> segment for the duration of the connection. If a non-<RST> segment is received without a TSopt, a TCP MAY drop the segment and send an <ACK> for the last in-sequence segment. A TCP MUST NOT abort a TCP connection if a non-<RST> segment is received without a TSopt.

If a TSopt is received on a connection where TSopt was not negotiated in the initial three-way handshake, the TSopt MUST be ignored and the packet processed normally.

In the case of crossing <SYN> segments where one <SYN> contains a TSopt and the other doesn't, both sides MAY send a TSopt in the <SYN,ACK> segment.

TSopt is required for the two mechanisms described in sections 3.3 and 4.2. There are also other mechanisms that rely on the presence of the TSopt, e.g. [RFC3522]. If a TCP stopped sending TSopt at any time during an established session, it interferes with these mechanisms. This update to [RFC1323] describes explicitly the previous assumption (see Section 4.2), that each TCP segment must have TSopt, once negotiated.

3.3. The RTTM Mechanism

RTTM places a Timestamp Option in every segment, with a TSval that is obtained from a (virtual) "timestamp clock". Values of this clock MUST be at least approximately proportional to real time, in order to measure actual RTT.

These TSval values are echoed in TSecr values in the reverse direction. The difference between a received TSecr value and the current timestamp clock value provides a RTT measurement.

When timestamps are used, every segment that is received will contain a TSecr value. However, these values cannot all be used to update the measured RTT. The following example illustrates why. It shows a one-way data flow with segments arriving in sequence without loss. Here A, B, C... represent data blocks occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding cumulative acknowledgments. The two timestamp fields of the Timestamp Option are shown symbolically as <TSval=x,TSecr=y>. Each TSecr field contains the value most recently received in a TSval field.

```
              TCP  A                                     TCP  B

                         <A,TSval=1,TSecr=120> ----->

                  <---- <ACK(A),TSval=127,TSecr=1>

                         <B,TSval=5,TSecr=127> ----->

                  <---- <ACK(B),TSval=131,TSecr=5>

               . . . . . . . . . . . . . . . . . . . . . .

                         <C,TSval=65,TSecr=131> ---->

                  <---- <ACK(C),TSval=191,TSecr=65>

                                (etc.)
```

The dotted line marks a pause (60 time units long) in which A had nothing to send. Note that this pause inflates the RTT which B could infer from receiving TSecr=131 in data segment C. Thus, in one-way data flows, RTTM in the reverse direction measures a value that is inflated by gaps in sending data. However, the following rule prevents a resulting inflation of the measured RTT:

RTTM Rule: A TSecr value received in a segment MAY be used to update the averaged RTT measurement only if the segment advances the left edge of the send window (e.g. SND.UNA is increased).

Since TCP B is not sending data, the data segment C does not acknowledge any new data when it arrives at B. Thus, the inflated RTTM measurement is not used to update B's RTTM measurement.

Implementers should note that with timestamps multiple RTTMs can be taken per RTT. Many RTO estimators have a weighting factor based on an implicit assumption that at most one RTTM will be sampled per RTT. When using multiple RTTMs per RTT to update the RTO estimator, the weighting factor needs to be decreased to take into account the more frequent RTTMs. For example, an implementation could choose to just use one sample per RTT to update the RTO estimator, or vary the gain based on the congestion window, or take an average of all the RTT measurements received over one RTT, and then use that value to update the RTO estimator. This document does not prescribe any particular method for modifying the RTO estimator.

3.4. Which Timestamp to Echo

If more than one Timestamp Option is received before a reply segment is sent, the TCP must choose only one of the TSvals to echo, ignoring the others. To minimize the state kept in the receiver (i.e., the number of unprocessed TSvals), the receiver should be required to retain at most one timestamp in the connection control block.

There are three situations to consider:

(A) Delayed ACKs.

Many TCP's acknowledge only every Kth segment out of a group of segments arriving within a short time interval; this policy is known generally as "delayed ACKs". The data-sender TCP must measure the effective RTT, including the additional time due to delayed ACKs, or else it will retransmit unnecessarily. Thus, when delayed ACKs are in use, the receiver SHOULD reply with the TSval field from the earliest unacknowledged segment.

(B) A hole in the sequence space (segment(s) have been lost).

The sender will continue sending until the window is filled, and the receiver may be generating <ACK>s as these out-of-order segments arrive (e.g., to aid "fast retransmit").

The lost segment is probably a sign of congestion, and in that situation the sender should be conservative about retransmission. Furthermore, it is better to overestimate than underestimate the RTT. An <ACK> for an out-of-order segment SHOULD therefore contain the timestamp from the most recent segment that advanced the window.

The same situation occurs if segments are re-ordered by the network.

(C) A filled hole in the sequence space.

The segment that fills the hole represents the most recent measurement of the network characteristics. A RTT computed from an earlier segment would probably include the sender's retransmit time-out, badly biasing the sender's average RTT estimate. Thus, the timestamp from the latest segment (which filled the hole) MUST be echoed.

An algorithm that covers all three cases is described in the following rules for Timestamp Option processing on a synchronized connection:

(1) The connection state is augmented with two 32-bit slots:

TS.Recent holds a timestamp to be echoed in TSecr whenever a segment is sent, and Last.ACK.sent holds the ACK field from the last segment sent. Last.ACK.sent will equal RCV.NXT except when <ACK>s have been delayed.

(2) If:

    SEG.TSval >= TS.recent and SEG.SEQ <= Last.ACK.sent

then SEG.TSval is copied to TS.Recent; otherwise, it is ignored.

(3) When a TSopt is sent, its TSecr field is set to the current TS.Recent value.

The following examples illustrate these rules. Here A, B, C... represent data segments occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding acknowledgment segments. Note that ACK(A) has the same sequence number as B. We show only one direction of timestamp echoing, for clarity.

o Segments arrive in sequence, and some of the <ACK>s are delayed.

   By case (A), the timestamp from the oldest unacknowledged segment is echoed.

```
                                         TS.Recent
   <A, TSval=1> ------------------->
                                             1
   <B, TSval=2> ------------------->
                                             1
   <C, TSval=3> ------------------->
                                             1
            <---- <ACK(C), TSecr=1>
   (etc)
```

o Segments arrive out of order, and every segment is acknowledged.

   By case (B), the timestamp from the last segment that advanced the left window edge is echoed, until the missing segment arrives; it is echoed according to Case (C). The same sequence would occur if segments B and D were lost and retransmitted.

```
                                                  TS.Recent
              <A, TSval=1> ------------------->
                                                      1
                       <---- <ACK(A), TSecr=1>
                                                      1
              <C, TSval=3> ------------------->
                                                      1
                       <---- <ACK(A), TSecr=1>
                                                      1
              <B, TSval=2> ------------------->
                                                      2
                       <---- <ACK(C), TSecr=2>
                                                      2
              <E, TSval=5> ------------------->
                                                      2
                       <---- <ACK(C), TSecr=2>
                                                      2
              <D, TSval=4> ------------------->
                                                      4
                       <---- <ACK(E), TSecr=4>
              (etc)
```

4. PAWS -- Protection Against Wrapped Sequence Numbers

4.1. Introduction

Section 4.2 describes a simple mechanism to reject old duplicate segments that might corrupt an open TCP connection; we call this mechanism PAWS (Protection Against Wrapped Sequence numbers). PAWS operates within a single TCP connection, using state that is saved in the connection control block. Section 4.8 and Appendix G discuss the implications of the PAWS mechanism for avoiding old duplicates from previous incarnations of the same connection.

4.2. The PAWS Mechanism

PAWS uses the same TCP Timestamp Option as the RTTM mechanism described earlier, and assumes that every received TCP segment (including data and <ACK> segments) contains a timestamp SEG.TSval whose values are monotonically non-decreasing in time. The basic idea is that a segment can be discarded as an old duplicate if it is received with a timestamp SEG.TSval less than some timestamp recently received on this connection.

In both the PAWS and the RTTM mechanism, the "timestamps" are 32-bit unsigned integers in a modular 32-bit space. Thus, "less than" is defined the same way it is for TCP sequence numbers, and the same

implementation techniques apply. If s and t are timestamp values,

$$s < t \text{ if } 0 < (t - s) < 2^{31},$$

computed in unsigned 32-bit arithmetic.

The choice of incoming timestamps to be saved for this comparison MUST guarantee a value that is monotonically increasing. For example, we might save the timestamp from the segment that last advanced the left edge of the receive window, i.e., the most recent in-sequence segment. Instead, we choose the value TS.Recent introduced in Section 3.4 for the RTTM mechanism, since using a common value for both PAWS and RTTM simplifies the implementation of both. As Section 3.4 explained, TS.Recent differs from the timestamp from the last in-sequence segment only in the case of delayed <ACK>s, and therefore by less than one window. Either choice will therefore protect against sequence number wrap-around.

RTTM was specified in a symmetrical manner, so that TSval timestamps are carried in both data and <ACK> segments and are echoed in TSecr fields carried in returning <ACK> or data segments. PAWS submits all incoming segments to the same test, and therefore protects against duplicate <ACK> segments as well as data segments. (An alternative non-symmetric algorithm would protect against old duplicate <ACK>s: the sender of data would reject incoming <ACK> segments whose TSecr values were less than the TSecr saved from the last segment whose ACK field advanced the left edge of the send window. This algorithm was deemed to lack economy of mechanism and symmetry.)

TSval timestamps sent on <SYN> and <SYN,ACK> segments are used to initialize PAWS. PAWS protects against old duplicate non-<SYN> segments, and duplicate <SYN> segments received while there is a synchronized connection. Duplicate <SYN> and <SYN,ACK> segments received when there is no connection will be discarded by the normal 3-way handshake and sequence number checks of TCP.

[RFC1323] recommended that <RST> segments NOT carry timestamps, and that they be acceptable regardless of their timestamp. At that time, the thinking was that old duplicate <RST> segments should be exceedingly unlikely, and their cleanup function should take precedence over timestamps. More recently, discussions about various blind attacks on TCP connections have raised the suggestion that if the timestamp option is present, SEG.TSecr could be used to provide stricter acceptance tests for <RST> segments. While still under discussion, to enable research into this area it is now RECOMMENDED that when generating a <RST>, that if the segment causing the <RST> to be generated contained a timestamp option, that the <RST> also contain a timestamp option. In the <RST> segment, SEG.TSecr SHOULD

be set to SEG.TSval from the incoming segment and SEG.TSval SHOULD be set to zero. If a <RST> is being generated because of a user abort, and Snd.TS.OK is set, then a timestamp option SHOULD be included in the <RST>. When a <RST> segment is received, it MUST NOT be subjected to PAWS checks, and information from the timestamp option MUST NOT be used to update connection state information. SEG.TSecr MAY be used to provide stricter <RST> acceptance checks.

4.3. Basic PAWS Algorithm

The PAWS algorithm REQUIRES the following processing to be performed on all incoming segments for a synchronized connection. Also, PAWS processing MUST take precedence over the regular TCP acceptablitiy check (Section 3.3 in [RFC0793]), which is performed after verification of the received timestamp option:

R1) If there is a Timestamp Option in the arriving segment, SEG.TSval < TS.Recent, TS.Recent is valid (see later discussion) and the RST bit is not set, then treat the arriving segment as not acceptable:

  Send an acknowledgement in reply as specified in [RFC0793] page 69 and drop the segment.

  Note: it is necessary to send an <ACK> segment in order to retain TCP's mechanisms for detecting and recovering from half-open connections. For example, see Figure 10 of [RFC0793].

R2) If the segment is outside the window, reject it (normal TCP processing)

R3) If an arriving segment satisfies: SEG.SEQ <= Last.ACK.sent (see Section 3.4), then record its timestamp in TS.Recent.

R4) If an arriving segment is in-sequence (i.e., at the left window edge), then accept it normally.

R5) Otherwise, treat the segment as a normal in-window, out-of-sequence TCP segment (e.g., queue it for later delivery to the user).

Steps R2, R4, and R5 are the normal TCP processing steps specified by [RFC0793].

It is important to note that the timestamp MUST be checked only when a segment first arrives at the receiver, regardless of whether it is in-sequence or it must be queued for later delivery.

Consider the following example.

Suppose the segment sequence: A.1, B.1, C.1, ..., Z.1 has been sent, where the letter indicates the sequence number and the digit represents the timestamp. Suppose also that segment B.1 has been lost. The timestamp in TS.Recent is 1 (from A.1), so C.1, ..., Z.1 are considered acceptable and are queued. When B is retransmitted as segment B.2 (using the latest timestamp), it fills the hole and causes all the segments through Z to be acknowledged and passed to the user. The timestamps of the queued segments are *not* inspected again at this time, since they have already been accepted. When B.2 is accepted, TS.Recent is set to 2.

This rule allows reasonable performance under loss. A full window of data is in transit at all times, and after a loss a full window less one segment will show up out-of-sequence to be queued at the receiver (e.g., up to ~2^30 bytes of data); the timestamp option must not result in discarding this data.

In certain unlikely circumstances, the algorithm of rules R1-R5 could lead to discarding some segments unnecessarily, as shown in the following example:

Suppose again that segments: A.1, B.1, C.1, ..., Z.1 have been sent in sequence and that segment B.1 has been lost. Furthermore, suppose delivery of some of C.1, ... Z.1 is delayed until AFTER the retransmission B.2 arrives at the receiver. These delayed segments will be discarded unnecessarily when they do arrive, since their timestamps are now out of date.

This case is very unlikely to occur. If the retransmission was triggered by a timeout, some of the segments C.1, ... Z.1 must have been delayed longer than the RTO time. This is presumably an unlikely event, or there would be many spurious timeouts and retransmissions. If B's retransmission was triggered by the "fast retransmit" algorithm, i.e., by duplicate <ACK>s, then the queued segments that caused these <ACK>s must have been received already.

Even if a segment were delayed past the RTO, the Fast Retransmit mechanism [Jacobson90c] will cause the delayed segments to be retransmitted at the same time as B.2, avoiding an extra RTT and therefore causing a very small performance penalty.

We know of no case with a significant probability of occurrence in which timestamps will cause performance degradation by unnecessarily discarding segments.

4.4. Timestamp Clock

It is important to understand that the PAWS algorithm does not require clock synchronization between sender and receiver. The sender's timestamp clock is used to stamp the segments, and the sender uses the echoed timestamp to measure RTTs. However, the receiver treats the timestamp as simply a monotonically increasing serial number, without any necessary connection to its clock. From the receiver's viewpoint, the timestamp is acting as a logical extension of the high-order bits of the sequence number.

The receiver algorithm does place some requirements on the frequency of the timestamp clock.

(a) The timestamp clock must not be "too slow".

It MUST tick at least once for each 2^31 bytes sent. In fact, in order to be useful to the sender for round trip timing, the clock SHOULD tick at least once per window's worth of data, and even with the window extension defined in Section 2.2, 2^31 bytes must be at least two windows.

To make this more quantitative, any clock faster than 1 tick/sec will reject old duplicate segments for link speeds of ~8 Gbps. A 1 ms timestamp clock will work at link speeds up to 8 Tbps (8*10^12) bps!

(b) The timestamp clock must not be "too fast".

The recycling time of the timestamp clock MUST be greater than MSL seconds. Since the clock (timestamp) is 32 bits and the worst-case MSL is 255 seconds, the maximum acceptable clock frequency is one tick every 59 ns.

However, it is desirable to establish a much longer recycle period, in order to handle outdated timestamps on idle connections (see Section 4.5), and to relax the MSL requirement for preventing sequence number wrap-around. With a 1 ms timestamp clock, the 32-bit timestamp will wrap its sign bit in 24.8 days. Thus, it will reject old duplicates on the same connection if MSL is 24.8 days or less. This appears to be a very safe figure; an MSL of 24.8 days or longer can probably be assumed in the internet without requiring precise MSL enforcement.

Based upon these considerations, we choose a timestamp clock frequency in the range 1 ms to 1 sec per tick. This range also matches the requirements of the RTTM mechanism, which does not need

much more resolution than the granularity of the retransmit timer, e.g., tens or hundreds of milliseconds.

The PAWS mechanism also puts a strong monotonicity requirement on the sender's timestamp clock. The method of implementation of the timestamp clock to meet this requirement depends upon the system hardware and software.

o Some hosts have a hardware clock that is guaranteed to be monotonic between hardware resets.

o A clock interrupt may be used to simply increment a binary integer by 1 periodically.

o The timestamp clock may be derived from a system clock that is subject to being abruptly changed, by adding a variable offset value. This offset is initialized to zero. When a new timestamp clock value is needed, the offset can be adjusted as necessary to make the new value equal to or larger than the previous value (which was saved for this purpose).

4.5. Outdated Timestamps

If a connection remains idle long enough for the timestamp clock of the other TCP to wrap its sign bit, then the value saved in TS.Recent will become too old; as a result, the PAWS mechanism will cause all subsequent segments to be rejected, freezing the connection (until the timestamp clock wraps its sign bit again).

With the chosen range of timestamp clock frequencies (1 sec to 1 ms), the time to wrap the sign bit will be between 24.8 days and 24800 days. A TCP connection that is idle for more than 24 days and then comes to life is exceedingly unusual. However, it is undesirable in principle to place any limitation on TCP connection lifetimes.

We therefore require that an implementation of PAWS include a mechanism to "invalidate" the TS.Recent value when a connection is idle for more than 24 days. (An alternative solution to the problem of outdated timestamps would be to send keep-alive segments at a very low rate, but still more often than the wrap-around time for timestamps, e.g., once a day. This would impose negligible overhead. However, the TCP specification has never included keep-alives, so the solution based upon invalidation was chosen.)

Note that a TCP does not know the frequency, and therefore, the wraparound time, of the other TCP, so it must assume the worst. The validity of TS.Recent needs to be checked only if the basic PAWS timestamp check fails, i.e., only if SEG.TSval < TS.Recent. If

TS.Recent is found to be invalid, then the segment is accepted, regardless of the failure of the timestamp check, and rule R3 updates TS.Recent with the TSval from the new segment.

To detect how long the connection has been idle, the TCP MAY update a clock or timestamp value associated with the connection whenever TS.Recent is updated, for example. The details will be implementation-dependent.

4.6. Header Prediction

"Header prediction" [Jacobson90a] is a high-performance transport protocol implementation technique that is most important for high-speed links. This technique optimizes the code for the most common case, receiving a segment correctly and in order. Using header prediction, the receiver asks the question, "Is this segment the next in sequence?" This question can be answered in fewer machine instructions than the question, "Is this segment within the window?"

Adding header prediction to our timestamp procedure leads to the following recommended sequence for processing an arriving TCP segment:

H1) Check timestamp (same as step R1 above)

H2) Do header prediction: if segment is next in sequence and if there are no special conditions requiring additional processing, accept the segment, record its timestamp, and skip H3.

H3) Process the segment normally, as specified in RFC 793. This includes dropping segments that are outside the window and possibly sending acknowledgments, and queuing in-window, out-of-sequence segments.

Another possibility would be to interchange steps H1 and H2, i.e., to perform the header prediction step H2 first, and perform H1 and H3 only when header prediction fails. This could be a performance improvement, since the timestamp check in step H1 is very unlikely to fail, and it requires unsigned modulo arithmetic. To perform this check on every single segment is contrary to the philosophy of header prediction. We believe that this change might produce a measurable reduction in CPU time for TCP protocol processing on high-speed networks.

However, putting H2 first would create a hazard: a segment from 2^32 bytes in the past might arrive at exactly the wrong time and be accepted mistakenly by the header-prediction step. The following reasoning has been introduced in [RFC1185] to show that the

probability of this failure is negligible.

If all segments are equally likely to show up as old duplicates, then the probability of an old duplicate exactly matching the left window edge is the maximum segment size (MSS) divided by the size of the sequence space. This ratio must be less than 2^-16, since MSS must be < 2^16; for example, it will be (2^12)/(2^32) = 2^-20 for a 100 Mbit/s link. However, the older a segment is, the less likely it is to be retained in the Internet, and under any reasonable model of segment lifetime the probability of an old duplicate exactly at the left window edge must be much smaller than 2^-16.

The 16 bit TCP checksum also allows a basic unreliability of one part in 2^16. A protocol mechanism whose reliability exceeds the reliability of the TCP checksum should be considered "good enough", i.e., it won't contribute significantly to the overall error rate. We therefore believe we can ignore the problem of an old duplicate being accepted by doing header prediction before checking the timestamp.

However, this probabilistic argument is not universally accepted, and the consensus at present is that the performance gain does not justify the hazard in the general case. It is therefore recommended that H2 follow H1.

4.7. IP Fragmentation

At high data rates, the protection against old segments provided by PAWS can be circumvented by errors in IP fragment reassembly (see [RFC4963]). The only way to protect against incorrect IP fragment reassembly is to not allow the segments to be fragmented. This is done by setting the Don't Fragment (DF) bit in the IP header. Setting the DF bit implies the use of Path MTU Discovery as described in [RFC1191], [RFC1981], and [RFC4821], thus any TCP implementation that implements PAWS MUST also implement Path MTU Discovery.

4.8. Duplicates from Earlier Incarnations of Connection

The PAWS mechanism protects against errors due to sequence number wrap-around on high-speed connections. Segments from an earlier incarnation of the same connection are also a potential cause of old duplicate errors. In both cases, the TCP mechanisms to prevent such errors depend upon the enforcement of a maximum segment lifetime (MSL) by the Internet (IP) layer (see Appendix of RFC 1185 for a detailed discussion). Unlike the case of sequence space wrap-around, the MSL required to prevent old duplicate errors from earlier incarnations does not depend upon the transfer rate. If the IP layer

enforces the recommended 2 minute MSL of TCP, and if the TCP rules are followed, TCP connections will be safe from earlier incarnations, no matter how high the network speed. Thus, the PAWS mechanism is not required for this case.

We may still ask whether the PAWS mechanism can provide additional security against old duplicates from earlier connections, allowing us to relax the enforcement of MSL by the IP layer. Appendix B explores this question, showing that further assumptions and/or mechanisms are required, beyond those of PAWS. This is not part of the current extension.

5. Conclusions and Acknowledgements

This memo presented a set of extensions to TCP to provide efficient operation over large bandwidth * delay product paths and reliable operation over very high-speed paths. These extensions are designed to provide compatible interworking with TCP stacks that do not implement the extensions.

These mechanisms are implemented using TCP options for scaled windows and timestamps. The timestamps are used for two distinct mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protection Against Wrapped Sequences).

The Window Scale option was originally suggested by Mike St. Johns of USAF/DCA. The present form of the option was suggested by Mike Karels of UC Berkeley in response to a more cumbersome scheme defined by Van Jacobson. Lixia Zhang helped formulate the PAWS mechanism description in [RFC1185].

Finally, much of this work originated as the result of discussions within the End-to-End Task Force on the theoretical limitations of transport protocols in general and TCP in particular. Task force members and other on the end2end-interest list have made valuable contributions by pointing out flaws in the algorithms and the documentation. Continued discussion and development since the publication of [RFC1323] originally occurred in the IETF TCP Large Windows Working Group, later on in the End-to-End Task Force, and most recently in the IETF TCP Maintenance Working Group. The authors are grateful for all these contributions.

6. Security Considerations

The TCP sequence space is a fixed size, and as the window becomes larger it becomes easier for an attacker to generate forged packets

that can fall within the TCP window, and be accepted as valid segments. While use of timestamps and PAWS can help to mitigate this, when using PAWS, if an attacker is able to forge a packet that is acceptable to the TCP connection, a timestamp that is in the future would cause valid segments to be dropped due to PAWS checks. Hence, implementers should take care to not open the TCP window drastically beyond the requirements of the connection.

Middle boxes and options: If a middle box removes TCP options from the <SYN> segment, such as TSopt, a high speed connection that needs PAWS would not have that protection. In this situation, an implementer could provide a mechanism for the application to determine whether or not PAWS is in use on the connection, and chose to terminate the connection if that protection doesn't exist.

Mechanisms to protect the TCP header from modification should also protect the TCP options.

A naive implementation that derives the timestamp clock value directly from a system uptime clock may unintentionally leak this information to an attacker. This does not directly compromise any of the mechanisms described in this document. However, this may be valuable information to a potential attacker. An implementer should evaluate the potential impact and mitigate this accordingly (i.e. by using a random offset for the timestamp clock on each connection, or using an external, real-time derived timestamp clock source).

Expanding the TCP window beyond 64K for IPv6 allows Jumbograms [RFC2675] to be used when the local network supports packets larger than 64K. When larger TCP segments are used, the TCP checksum becomes weaker.

7. IANA Considerations

This document has no actions for IANA.

8. References

8.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

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

8.2. Informative References

[Garlick77]
Garlick, L., Rom, R., and J. Postel, "Issues in Reliable Host-to-Host Protocols", Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, <http://www.rfc-editor.org/ien/ien12.txt>.

[Hamming77]
Hamming, R., "Digital Filters", Prentice Hall, Englewood Cliffs, N.J. ISBN 0-13-212571-4, 1977.

[Jacobson88a]
Jacobson, V., "Congestion Avoidance and Control", SIGCOMM '88, Stanford, CA., August 1988, <http://ee.lbl.gov/papers/congavoid.pdf>.

[Jacobson90a]
Jacobson, V., "4BSD Header Prediction", ACM Computer Communication Review, April 1990.

[Jacobson90c]
Jacobson, V., "Modified TCP congestion avoidance algorithm", Message to the end2end-interest mailing list, April 1990, <ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail>.

[Jain86] Jain, R., "Divergence of Timeout Algorithms for Packet Retransmissions", Proc. Fifth Phoenix Conf. on Comp. and Comm., Scottsdale, Arizona, March 1986, <http://arxiv.org/ftp/cs/papers/9809/9809097.pdf>.

[Karn87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, August 1987.

[Martin03]
Martin, D., "[Tsvwg] RFC 1323.bis", Message to the tsvwg mailing list, September 2003, <http://www.ietf.org/mail-archive/web/tsvwg/current/msg04435.html>.

[Mathis08]
Mathis, M., "[tcpm] Example of 1323 window retraction problem", Message to the tcpm mailing list, March 2008, <http://www.ietf.org/mail-archive/web/tcpm/current/

msg03564.html>.

[RFC0896] Nagle, J., "Congestion control in IP/TCP internetworks", RFC 896, January 1984.

[RFC1072] Jacobson, V. and R. Braden, "TCP extensions for long-delay paths", RFC 1072, October 1988.

[RFC1110] McKenzie, A., "Problem with the TCP big window option", RFC 1110, August 1989.

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

[RFC1185] Jacobson, V., Braden, B., and L. Zhang, "TCP Extension for High-Speed Paths", RFC 1185, October 1990.

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

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

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

[RFC2581] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, August 1999.

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, July 2007.

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

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

[RFC6691] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, July 2012.

[Watson81] Watson, R., "Timer-based Mechanisms in Reliable Transport Protocol Connection Management", Computer Networks, Vol. 5, 1981.

[Zhang86] Zhang, L., "Why TCP Timers Don't Work Well", Proc. SIGCOMM '86, Stowe, VT, August 1986.

Appendix A. Implementation Suggestions

TCP Option Layout

The following layouts are recommended for sending options on non-<SYN> segments, to achieve maximum feasible alignment of 32-bit and 64-bit machines.

```
+--------+--------+--------+--------+
|   NOP  |  NOP   |  TSopt |   10   |
+--------+--------+--------+--------+
|          TSval timestamp          |
+--------+--------+--------+--------+
|          TSecr timestamp          |
+--------+--------+--------+--------+
```

Interaction with the TCP Urgent Pointer

The TCP Urgent pointer, like the TCP window, is a 16 bit value. Some of the original discussion for the TCP Window Scale option included proposals to increase the Urgent pointer to 32 bits. As it turns out, this is unnecessary. There are two observations that should be made:

(1) With IP Version 4, the largest amount of TCP data that can be sent in a single packet is 65495 bytes (64K - 1 -- size of fixed IP and TCP headers).

(2) Updates to the urgent pointer while the user is in "urgent mode" are invisible to the user.

This means that if the Urgent Pointer points beyond the end of the TCP data in the current segment, then the user will remain in urgent mode until the next TCP segment arrives. That segment will update the urgent pointer to a new offset, and the user will never have left urgent mode.

Thus, to properly implement the Urgent Pointer, the sending TCP only has to check for overflow of the 16 bit Urgent Pointer field before filling it in. If it does overflow, than a value of 65535 should be inserted into the Urgent Pointer.

The same technique applies to IP Version 6, except in the case of IPv6 Jumbograms. When IPv6 Jumbograms are supported, [RFC2675] requires additional steps for dealing with the Urgent Pointer, these are described in section 5.2 of [RFC2675].

Appendix B. Duplicates from Earlier Connection Incarnations

There are two cases to be considered: (1) a system crashing (and losing connection state) and restarting, and (2) the same connection being closed and reopened without a loss of host state. These will be described in the following two sections.

B.1. System Crash with Loss of State

TCP's quiet time of one MSL upon system startup handles the loss of connection state in a system crash/restart. For an explanation, see for example "When to Keep Quiet" in the TCP protocol specification [RFC0793]. The MSL that is required here does not depend upon the transfer speed. The current TCP MSL of 2 minutes seemed acceptable as an operational compromise, when many host systems used to take this long to boot after a crash. Current host systems can boot considerably faster.

The timestamp option may be used to ease the MSL requirements (or to provide additional security against data corruption). If timestamps are being used and if the timestamp clock can be guaranteed to be monotonic over a system crash/restart, i.e., if the first value of the sender's timestamp clock after a crash/restart can be guaranteed to be greater than the last value before the restart, then a quiet time is unnecessary.

To dispense totally with the quiet time would require that the host clock be synchronized to a time source that is stable over the crash/

restart period, with an accuracy of one timestamp clock tick or better. We can back off from this strict requirement to take advantage of approximate clock synchronization. Suppose that the clock is always re-synchronized to within N timestamp clock ticks and that booting (extended with a quiet time, if necessary) takes more than N ticks. This will guarantee monotonicity of the timestamps, which can then be used to reject old duplicates even without an enforced MSL.

B.2. Closing and Reopening a Connection

When a TCP connection is closed, a delay of 2*MSL in TIME-WAIT state ties up the socket pair for 4 minutes (see Section 3.5 of [RFC0793]. Applications built upon TCP that close one connection and open a new one (e.g., an FTP data transfer connection using Stream mode) must choose a new socket pair each time. The TIME-WAIT delay serves two different purposes:

(a) Implement the full-duplex reliable close handshake of TCP.

The proper time to delay the final close step is not really related to the MSL; it depends instead upon the RTO for the FIN segments and therefore upon the RTT of the path. (It could be argued that the side that is sending a FIN knows what degree of reliability it needs, and therefore it should be able to determine the length of the TIME-WAIT delay for the FIN's recipient. This could be accomplished with an appropriate TCP option in FIN segments.)

Although there is no formal upper-bound on RTT, common network engineering practice makes an RTT greater than 1 minute very unlikely. Thus, the 4 minute delay in TIME-WAIT state works satisfactorily to provide a reliable full-duplex TCP close. Note again that this is independent of MSL enforcement and network speed.

The TIME-WAIT state could cause an indirect performance problem if an application needed to repeatedly close one connection and open another at a very high frequency, since the number of available TCP ports on a host is less than 2^16. However, high network speeds are not the major contributor to this problem; the RTT is the limiting factor in how quickly connections can be opened and closed. Therefore, this problem will be no worse at high transfer speeds.

(b) Allow old duplicate segments to expire.

To replace this function of TIME-WAIT state, a mechanism would have to operate across connections. PAWS is defined strictly within a single connection; the last timestamp (TS.Recent) is kept in the connection control block, and discarded when a connection is closed.

An additional mechanism could be added to the TCP, a per-host cache of the last timestamp received from any connection. This value could then be used in the PAWS mechanism to reject old duplicate segments from earlier incarnations of the connection, if the timestamp clock can be guaranteed to have ticked at least once since the old connection was open. This would require that the TIME-WAIT delay plus the RTT together must be at least one tick of the sender's timestamp clock. Such an extension is not part of the proposal of this RFC.

Note that this is a variant on the mechanism proposed by Garlick, Rom, and Postel [Garlick77], which required each host to maintain connection records containing the highest sequence numbers on every connection. Using timestamps instead, it is only necessary to keep one quantity per remote host, regardless of the number of simultaneous connections to that host.

Appendix C. Summary of Notation

The following notation has been used in this document.

Options

```
WSopt:             TCP Window Scale Option
TSopt:             TCP Timestamp Option
```

Option Fields

```
shift.cnt:         Window scale byte in WSopt
TSval:             32-bit Timestamp Value field in TSopt
TSecr:             32-bit Timestamp Reply field in TSopt
```

Option Fields in Current Segment

```
SEG.TSval:         TSval field from TSopt in current segment
```

```
      SEG.TSecr:        TSecr field from TSopt in current segment
      SEG.WSopt:        8-bit value in WSopt
```

Clock Values

```
      my.TSclock:       System wide source of 32-bit timestamp values
      my.TSclock.rate:  Period of my.TSclock (1 ms to 1 sec)
      Snd.TSoffset:     A offset for randomizing Snd.TSclock
      Snd.TSclock:      my.TSclock + Snd.TSoffset
```

Per-Connection State Variables

```
      TS.Recent:        Latest received Timestamp
      Last.ACK.sent:    Last ACK field sent
      Snd.TS.OK:        1-bit flag
      Snd.WS.OK:        1-bit flag
      Rcv.Wind.Scale:   Receive window scale power
      Snd.Wind.Scale:   Send window scale power
      Start.Time:       Snd.TSclock value when segment being timed was
                        sent (used by pre-1323 code).
```

Procedure

```
      Update_SRTT(m)    Procedure to update the smoothed RTT and RTT
                        variance estimates, using the rules of
                        [Jacobson88a], given m, a new RTT measurement
```

Appendix D. Event Processing Summary

OPEN Call

...

An initial send sequence number (ISS) is selected. Send a <SYN> segment of the form:

```
        <SEQ=ISS><CTL=SYN><TSval=Snd.TSclock><WSopt=Rcv.Wind.Scale>
```

...

SEND Call

CLOSED STATE (i.e., TCB does not exist)

...

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a <SYN> segment containing the options: <TSval=Snd.TSclock> and <WSopt=Rcv.Wind.Scale>. Set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. ...

SYN-SENT STATE
SYN-RECEIVED STATE

...

ESTABLISHED STATE
CLOSE-WAIT STATE

Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). ...

If the urgent flag is set ...

If the Snd.TS.OK flag is set, then include the TCP Timestamp Option <TSval=Snd.TSclock,TSecr=TS.Recent> in each data segment.

Scale the receive window for transmission in the segment header:

```
SEG.WND = (RCV.WND >> Rcv.Wind.Scale).
```

SEGMENT ARRIVES

...

If the state is LISTEN then

first check for an RST

...

second check for an ACK

...

third check for a SYN

if the SYN bit is set, check the security. If the ...

...

if the SEG.PRC is less than the TCB.PRC then continue.

Check for a Window Scale option (WSopt); if one is found, save SEG.WSopt in Snd.Wind.Scale and set Snd.WS.OK flag on. Otherwise, set both Snd.Wind.Scale and Rcv.Wind.Scale to zero and clear Snd.WS.OK flag.

Check for a TSopt option; if one is found, save SEG.TSval in the variable TS.Recent and turn on the Snd.TS.OK bit.

Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any other control or text should be queued for processing later. ISS should be selected and a <SYN> segment sent of the form:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

If the Snd.WS.OK bit is on, include a WSopt option <WSopt=Rcv.Wind.Scale> in this segment. If the Snd.TS.OK bit is on, include a TSopt <TSval=Snd.TSclock,TSecr=TS.Recent> in this segment. Last.ACK.sent is set to RCV.NXT.

SND.NXT is set to ISS+1 and SND.UNA to ISS. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control or data (combined with SYN) will be processed in the SYN-RECEIVED state, but processing of SYN and ACK should not be repeated. If the listen was not fully specified (i.e., the foreign socket was not fully specified), then the unspecified fields should be filled in now.

fourth other text or control

...

If the state is SYN-SENT then

first check the ACK bit

...

...

fourth check the SYN bit

...

If the SYN bit is on and the security/compartment and precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ, and any acknowledgements on the retransmission queue which are thereby acknowledged should be removed.

Check for a Window Scale option (WSopt); if it is found, save SEG.WSopt in Snd.Wind.Scale; otherwise, set both Snd.Wind.Scale and Rcv.Wind.Scale to zero.

Check for a TSopt option; if one is found, save SEG.TSval in variable TS.Recent and turn on the Snd.TS.OK bit in the connection control block. If the ACK bit is set, use Snd.TSclock - SEG.TSecr as the initial RTT estimate.

If SND.UNA > ISS (our <SYN> has been ACKed), change the connection state to ESTABLISHED, form an <ACK> segment:

```
        <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
```

and send it. If the Snd.Echo.OK bit is on, include a TSopt option <TSval=Snd.TSclock,TSecr=TS.Recent> in this <ACK> segment. Last.ACK.sent is set to RCV.NXT.

Data or controls which were queued for transmission may be included. If there are other controls or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

Otherwise enter SYN-RECEIVED, form a <SYN,ACK> segment:

```
        <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
```

and send it. If the Snd.Echo.OK bit is on, include a TSopt option <TSval=Snd.TSclock,TSecr=TS.Recent> in this segment. If the Snd.WS.OK bit is on, include a WSopt option <WSopt=Rcv.Wind.Scale> in this segment. Last.ACK.sent is set to RCV.NXT.

If there are other controls or text in the segment, queue them for processing after the ESTABLISHED state has been reached, return.

fifth, if neither of the SYN or RST bits is set then drop the segment and return.

```
      Otherwise,

      First, check sequence number

         SYN-RECEIVED STATE
         ESTABLISHED STATE
         FIN-WAIT-1 STATE
         FIN-WAIT-2 STATE
         CLOSE-WAIT STATE
         CLOSING STATE
         LAST-ACK STATE
         TIME-WAIT STATE

            Segments are processed in sequence.  Initial tests on
            arrival are used to discard old duplicates, but further
            processing is done in SEG.SEQ order.  If a segment's
            contents straddle the boundary between old and new, only the
            new parts should be processed.

            Rescale the received window field:

                  TrueWindow = SEG.WND << Snd.Wind.Scale,

            and use "TrueWindow" in place of SEG.WND in the following
            steps.

            Check whether the segment contains a Timestamp Option and
            bit Snd.TS.OK is on.  If so:

               If SEG.TSval < TS.Recent and the RST bit is off, then
               test whether connection has been idle less than 24 days;
               if all are true, then the segment is not acceptable;
               follow steps below for an unacceptable segment.

               If SEG.SEQ is less than or equal to Last.ACK.sent, then
               save SEG.TSval in variable TS.Recent.

            There are four cases for the acceptability test for an
            incoming segment:

               ...

            If an incoming segment is not acceptable, an acknowledgment
            should be sent in reply (unless the RST bit is set, if so
            drop the segment and return):

                    <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
```

Last.ACK.sent is set to SEG.ACK of the acknowledgment. If the Snd.Echo.OK bit is on, include the Timestamp Option <TSval=Snd.TSclock,TSecr=TS.Recent> in this <ACK> segment. Set Last.ACK.sent to SEG.ACK and send the <ACK> segment. After sending the acknowledgment, drop the unacceptable segment and return.

...

fifth check the ACK field.

if the ACK bit is off drop the segment and return.

if the ACK bit is on

...

ESTABLISHED STATE

If SND.UNA < SEG.ACK <= SND.NXT then, set SND.UNA <- SEG.ACK. Also compute a new estimate of round-trip time. If Snd.TS.OK bit is on, use Snd.TSclock - SEG.TSecr; otherwise use the elapsed time since the first segment in the retransmission queue was sent. Any segments on the retransmission queue which are thereby entirely acknowledged...

...

Seventh, process the segment text.

ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE

...

Send an acknowledgment of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

If the Snd.TS.OK bit is on, include Timestamp Option <TSval=Snd.TSclock,TSecr=TS.Recent> in this <ACK> segment. Set Last.ACK.sent to SEG.ACK of the acknowledgment, and send it. This acknowledgment should be piggy-backed on a segment being transmitted if possible without incurring undue delay.

```
         ...
```

Appendix E. Timestamps Edge Cases

While the rules laid out for when to calculate RTTM produce the correct results most of the time, there are some edge cases where an incorrect RTTM can be calculated. All of these situations involve the loss of segments. It is felt that these scenarios are rare, and that if they should happen, they will cause a single RTTM measurement to be inflated, which mitigates its effects on RTO calculations.

[Martin03] cites two similar cases when the returning <ACK> is lost, and before the retransmission timer fires, another returning <ACK> segment arrives, which aknowledges the data. In this case, the RTTM calculated will be inflated:

```
        clock
          tc=1   <A, TSval=1> ------------------->

          tc=2   (lost) <---- <ACK(A), TSecr=1, win=n>
              (RTTM would have been 1)

                 (receive window opens, window update is sent)
          tc=5        <---- <ACK(A), TSecr=1, win=m>
                 (RTTM is calculated at 4)
```

One thing to note about this situation is that it is somewhat bounded by RTO + RTT, limiting how far off the RTTM calculation will be. While more complex scenarios can be constructed that produce larger inflations (e.g., retransmissions are lost), those scenarios involve multiple segment losses, and the connection will have other more serious operational problems than using an inflated RTTM in the RTO calculation.

Appendix F. Window Retraction Example

Consider a established TCP connection with WSCALE=7 (128 byte receiver window quantization), that is running with a very small windows because the receiver is bottlenecked and both ends are doing small reads and writes.

Consider the ACKs coming back:

```
SEG.ACK  SEG.WIN computed SND.WIN   receiver's actual window
1000     2       1256               1300
```

The sender writes 40 bytes and receiver ACKs:

```
1040     2       1296               1300
```

The sender writes 5 additional bytes and the receiver has a problem. Two choices:

```
1045     2       1301               1300   - BEYOND BUFFER

1045     1       1173               1300   - RETRACTED WINDOW
```

This problems is completely general and can in principle happen any time the sender does a write which is smaller than the window scale quanta.

In most stacks it is at least partially obscured when the window size is larger than some small number of segments because the stacks prefer to announce windows that are integral numbers of segments (rounded up to the next window quanta). This plus silly window suppression tends to cause less frequent, larger window updates. If the window was rounded down to a segment size there is more opportunity to advance it ("beyond buffer" case above) rather than retracting it.

Appendix G. Changes from RFC 1323

Several important updates and clarifications to the specification in RFC 1323 are made in these document. The technical changes are summarized below:

(a) Section 2.4 was added describing the unavoidable window retraction issue, and explicitly describing the mitigation steps necessary.

(b) In Section 3.2 the wording how timestamp option negotiation is to be performed was updated with RFC2119 wording. Further, a number of paragraphs were added to clarify the expected behavior with a compliant implementation using TSopt, as RFC1323 left room for interpretation - e.g. potential late enablement of TSopt.

(c) The description of which TSecr values can be used to update the measured RTT has been clarified. Specifically, with timestamps, the Karn algorithm [Karn87] is disabled. The Karn algorithm disables all RTT measurements during retransmission, since it is ambiguous whether the <ACK> is for the original segment, or the retransmitted segment. With timestamps, that ambiguity is

removed since the TSecr in the <ACK> will contain the TSval from whichever data segment made it to the destination.

(d) RTTM update processing explicitly excludes segments not updating SND.UNA. The original text could be interpreted to allow taking RTT samples when SACK acknowledges some new, non-continuous data.

(e) In RFC1323, section 3.4, step (2) of the algorithm to control which timestamp is echoed was incorrect in two regards:

(1) It failed to update TS.recent for a retransmitted segment that resulted from a lost <ACK>.

(2) It failed if SEG.LEN = 0.

In the new algorithm, the case of SEG.TSval >= TS.recent is included for consistency with the PAWS test.

(f) It is now recommended that Timestamp Options be included in <RST> segments if the incoming segment contained a Timestamp Option.

(g) <RST> segments are explicitly excluded from PAWS processing.

(h) Added text to clarify the precedence between regular TCP [RFC0793] and timestamp/PAWS [RFCxxxx] processing. Discussion about combined acceptability checks are ongoing.

(i) Snd.TSoffset and Snd.TSclock variables have been added. Snd.TSclock is the sum of my.TSclock and Snd.TSoffset. This allows the starting points for timestamp values to be randomized on a per-connection basis. Setting Snd.TSoffset to zero yields the same results as [RFC1323].

(j) Appendix A has been expanded with information about the TCP Urgent Pointer. An earlier revision contained text around the TCP MSS option, which was split off into [RFC6691].

(k) One correction was made to the Event Processing Summary in Appendix D. In SEND CALL/ESTABLISHED STATE, RCV.WND is used to fill in the SEG.WND value, not SND.WND.

Editorial changes of the document, that don't impact the implementation or function of the mechanisms described in this document include:

(a) Removed much of the discussion in Section 1 to streamline the document. However, detailed examples and discussions in Section 2, Section 3 and Section 4 are kept as guideline for implementers.

(b) Removed references to "new" options, as the options were introduced in [RFC1323] already. Changed the text in Section 1.3 to specifically address TS and WS options.

(c) Section 1.4 was added for RFC2119 wording. Normative text was updated with the appropriate phrases.

(d) Added < > brackets to mark specific types of segments, and replaced most occurances of "packet" with "segment", where TCP segments are referred.

(e) Removed the list of changes between RFC 1323 and prior versions. These changes are mentioned in appendix C of RFC 1323.

(f) Moved Appendix "Changes" at the end of the appendices for easier lookup. In addition, the entries were split into a technical and an editorial part, and sorted to roughly correspond with the sections in the text where they apply.

Authors' Addresses

David Borman
Quantum Corporation
Mendota Heights MN 55120
USA

Email: david.borman@quantum.com

Bob Braden
University of Southern California
4676 Admiralty Way
Marina del Rey CA 90292
USA

Email: braden@isi.edu

Van Jacobson
Packet Design
2465 Latham Street
Mountain View CA 94040
USA

Email: van@packetdesign.com

Richard Scheffenegger (editor)
NetApp, Inc.
Am Euro Platz 2
Vienna,   1120
Austria

Email: rs@netapp.com