Network Working Group   V. Jacobson
Request for Comments: 1323   LBL
Obsoletes: RFC 1072, RFC 1185   R. Braden
                                                                 ISI
                                                           D. Borman
                                                       Cray Research
                                                            May 1992

TCP Extensions for High Performance

Status of This Memo

Abstract

   This memo presents a set of TCP extensions to improve performance
   over large bandwidth*delay product paths and to provide reliable
   operation over very high-speed paths. It defines new TCP options for
   scaled windows and timestamps, which are designed to provide
   compatible interworking with TCP's that do not implement the
   extensions. The timestamps are used for two distinct mechanisms:
   RTTM (Round Trip Time Measurement) and PAWS (Protect Against Wrapped
   Sequences). Selective acknowledgments are not included in this memo.

   This memo combines and supersedes RFC-1072 and RFC-1185, adding
   additional clarification and more detailed specification. Appendix C
   summarizes the changes from the earlier RFCs.

TABLE OF CONTENTS

1. INTRODUCTION

   The TCP protocol [Postel81] was designed to operate reliably over
   almost any transmission medium regardless of transmission rate,
   delay, corruption, duplication, or reordering of segments.
   Production TCP implementations currently adapt to transfer rates in
   the range of 100 bps to 10**7 bps and round-trip delays in the range
   1 ms to 100 seconds. Recent work on TCP performance has shown that
   TCP can work well over a variety of Internet paths, ranging from 800
   Mbit/sec I/O channels to 300 bit/sec dial-up modems [Jacobson88a].

   The introduction of fiber optics is resulting in ever-higher
   transmission speeds, and the fastest paths are moving out of the
   domain for which TCP was originally engineered. This memo defines a
   set of modest extensions to TCP to extend the domain of its
   application to match this increasing network capability. It is based
   upon and obsoletes RFC-1072 [Jacobson88b] and RFC-1185 [Jacobson90b].

   There is no one-line answer to the question: "How fast can TCP go?".
   There are two separate kinds of issues, performance and reliability,
   and each depends upon different parameters. We discuss each in turn.

   1.1 TCP Performance

      TCP performance depends not upon the transfer rate itself, but
      rather upon the product of the transfer rate and the round-trip
      delay. This "bandwidth*delay product" measures the amount of data
      that would "fill the pipe"; it is the buffer space required at
      sender and receiver to obtain maximum throughput on the TCP
      connection over the path, i.e., the amount of unacknowledged data
      that TCP must handle in order to keep the pipeline full. TCP
      performance problems arise when the bandwidth*delay product is
      large. We refer to an Internet path operating in this region as a
      "long, fat pipe", and a network containing this path as an "LFN"
      (pronounced "elephan(t)").

      High-capacity packet satellite channels (e.g., DARPA's Wideband
      Net) are LFN's. For example, a DS1-speed satellite channel has a
      bandwidth*delay product of 10**6 bits or more; this corresponds to
      100 outstanding TCP segments of 1200 bytes each. Terrestrial
      fiber-optical paths will also fall into the LFN class; for
      example, a cross-country delay of 30 ms at a DS3 bandwidth
      (45Mbps) also exceeds 10**6 bits.

      There are three fundamental performance problems with the current
      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
new 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. Until recently, properly-operating TCP
implementations would cause the data pipeline to drain with
every packet loss, and require a slow-start action to
recover. Recently, the Fast Retransmit and Fast Recovery
algorithms [Jacobson90c] have been introduced. Their
combined effect is to recover from one packet loss per
window, without draining the pipeline. However, more than
one packet loss per window typically results in a
retransmission timeout and the resulting pipeline drain and
slow start.

Expanding the window size to match the capacity of an LFN
results in a corresponding increase of the probability of
more than one packet per window being dropped. This could
have a devastating effect upon the throughput of TCP over an
LFN. In addition, if a congestion control mechanism based
upon some form of random dropping were introduced into
gateways, randomly spaced packet drops would become common,
possible increasing the probability of dropping more than one
packet per window.

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. Some evidence in
favor of selective acknowledgments has been published
[NBS85], and selective acknowledgments have been included in
a number of experimental Internet protocols -- VMTP
[Cheriton88], NETBLT [Clark87], and RDP [Velten84], and
proposed for OSI TP4 [NBS85]. However, in the non-LFN
regime, selective acknowledgments reduce the number of

packets retransmitted but do not otherwise improve
performance, making their complexity of questionable value.
However, selective acknowledgments are expected to become
much more important in the LFN regime.

RFC-1072 defined a new TCP "SACK" option to send a selective
acknowledgment. However, there are important technical
issues to be worked out concerning both the format and
semantics of the SACK option. Therefore, SACK has been
omitted from this package of extensions. It is hoped that
SACK can "catch up" during the standardization process.

(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 4 introduces a new TCP option, "Timestamps", and then
defines 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 Timestamps option.

1.2 TCP Reliability

Now we turn from performance to reliability. High transfer rate
enters TCP performance through the bandwidth*delay product.
However, high transfer rate alone can threaten TCP reliability by
violating the assumptions behind the TCP mechanism for duplicate
detection and sequencing.

An especially serious kind of error may result from an accidental
reuse of TCP sequence numbers in data segments. Suppose that an
"old duplicate segment", e.g., a duplicate data segment that was
delayed in Internet queues, is delivered to the receiver at the
wrong moment, so that its sequence numbers falls somewhere within
the current window. There would be no checksum failure to warn of
the error, and the result could be an undetected corruption of the
data. Reception of an old duplicate ACK segment at the
transmitter could be only slightly less serious: it is likely to

lock up the connection so that no further progress can be made, forcing an RST on the connection.

TCP reliability depends upon the existence of a bound on the lifetime of a segment: the "Maximum Segment Lifetime" or MSL. An MSL is generally required by any reliable transport protocol, since every sequence number field must be finite, and therefore any sequence number may eventually be reused. In the Internet protocol suite, the MSL bound is enforced by an IP-layer mechanism, the "Time-to-Live" or TTL field.

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 sockets) 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 spec, as explained in Section 5.3 and Appendix B. However, case (1), avoiding the reuse of sequence numbers within the same connection, requires an MSL bound that depends upon the transfer rate, and at high enough rates, a new mechanism is required.

More specifically, if the maximum effective bandwidth at which TCP is able to transmit over a particular path is B bytes per second, then the following constraint must be satisfied for error-free operation:

$$2**31 / B > MSL (secs) \qquad\qquad [1]$$

The following table shows the value for Twrap = 2**31/B in seconds, for some important values of the bandwidth B:

Network	B*8 bits/sec	B bytes/sec	Twrap secs
ARPANET	56kbps	7KBps	3*10**5 (~3.6 days)
DS1	1.5Mbps	190KBps	10**4 (~3 hours)
Ethernet	10Mbps	1.25MBps	1700 (~30 mins)
DS3	45Mbps	5.6MBps	380
FDDI	100Mbps	12.5MBps	170
Gigabit	1Gbps	125MBps	17

It is clear that wrap-around of the sequence space is not a
problem for 56kbps packet switching or even 10Mbps Ethernets. On
the other hand, at DS3 and FDDI speeds, Twrap is comparable to the
2 minute MSL assumed by the TCP specification [Postel81]. Moving
towards gigabit speeds, Twrap becomes too small for reliable
enforcement by the Internet TTL mechanism.

The 16-bit window field of TCP limits the effective bandwidth B to
2**16/RTT, where RTT is the round-trip time in seconds
[McKenzie89]. If the RTT is large enough, this limits B to a
value that meets the constraint [1] for a large MSL value. For
example, consider a transcontinental backbone with an RTT of 60ms
(set by the laws of physics). With the bandwidth*delay product
limited to 64KB by the TCP window size, B is then limited to
1.1MBps, no matter how high the theoretical transfer rate of the
path. This corresponds to cycling the sequence number space in
Twrap= 2000 secs, which is safe in today's Internet.

It is important to understand that the culprit is not the larger
window but rather the high bandwidth. For example, consider a
(very large) FDDI LAN with a diameter of 10km. Using the speed of
light, we can compute the RTT across the ring as
(2*10**4)/(3*10**8) = 67 microseconds, and the delay*bandwidth
product is then 833 bytes. A TCP connection across this LAN using
a window of only 833 bytes will run at the full 100mbps and can
wrap the sequence space in about 3 minutes, very close to the MSL
of TCP. Thus, high speed alone can cause a reliability problem
with sequence number wrap-around, even without extended windows.

Watson's Delta-T protocol [Watson81] includes network-layer
mechanisms for precise enforcement of an MSL. In contrast, the IP

mechanism for MSL enforcement is loosely defined and even more
loosely implemented in the Internet. Therefore, it is unwise to
depend upon active enforcement of MSL for TCP connections, and it
is unrealistic to imagine setting MSL's smaller than the current
values (e.g., 120 seconds specified for TCP).

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

1.3 Using TCP options

The extensions defined in this memo all use new TCP options. We
must address two possible issues concerning the use of TCP
options: (1) compatibility and (2) overhead.

We must pay careful attention to compatibility, i.e., to
interoperation with existing implementations. The only TCP option
defined previously, MSS, may appear only on a SYN segment. Every
implementation should (and we expect that most will) ignore
unknown options on SYN segments. However, some buggy TCP
implementation might be crashed by the first appearance of an
option on a non-SYN segment. Therefore, for each of the
extensions defined below, TCP options will be sent on non-SYN
segments only when an exchange of options on the SYN segments has
indicated that both sides understand the extension. Furthermore,
an extension option will be sent in a <SYN,ACK> segment only if
the corresponding option was received in the initial <SYN>
segment.

A question may be raised about the bandwidth and processing
overhead for TCP options. Those options that occur on SYN
segments are not likely to cause a performance concern. Opening a
TCP connection requires execution of significant special-case
code, and the processing of options is unlikely to increase that
cost significantly.

On the other hand, a Timestamps option may appear in any data or
ACK segment, adding 12 bytes to the 20-byte TCP header. We

believe that the bandwidth saved by reducing unnecessary
retransmissions will more than pay for the extra header bandwidth.

There is also an issue about the processing overhead for parsing
the variable byte-aligned format of options, particularly with a
RISC-architecture CPU. To meet this concern, Appendix A contains
a recommended layout of the options in TCP headers to achieve
reasonable data field alignment. In the spirit of Header
Prediction, a TCP can quickly test for this layout and if it is
verified then use a fast path. Hosts that use this canonical
layout will effectively use the options as a set of fixed-format
fields appended to the TCP header. However, to retain the
philosophical and protocol framework of TCP options, a TCP must be
prepared to parse an arbitrary options field, albeit with less
efficiency.

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.

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 new 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. (Another design choice would be to
     specify the window scale in every TCP segment. It would be
     incorrect to send a window scale option only when the scale factor
     changed, since a TCP option in an acknowledgement segment will not
     be delivered reliably (unless the ACK happens to be piggy-backed
     on data in the other direction). Fixing the scale when the
     connection is opened has the advantage of lower overhead but the
     disadvantage that the scale factor cannot be changed during the
     connection.)

     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|
        +---------+---------+---------+
```


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 op-
tion 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 SYN (i.e., a <SYN> or <SYN,ACK>) segment
itself is never scaled.

2.3 Using the Window Scale Option

A model implementation of window scaling is as follows, using the
notation of RFC-793 [Postel81]:

*   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.

*   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.

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

*   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.

*   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.

*   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 RFC793 are met, and using
    the "C" notation "<<" for left-shift).

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

        SEG.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 2 * 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.

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" [Nagle84] 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 only one packet per window. While this yields an adequate
      approximation to the RTT for small windows, it results in an
      unacceptably poor RTT estimate for an 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
      packets, 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 packets, the RTT estimator may be seriously in error,
      resulting in spurious retransmissions.

      If there are dropped packets, 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 Timestamps 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 Timestamps Option.

TCP Timestamps Option (TSopt):

Kind: 8

Length: 10 bytes

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

The Timestamps 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 only valid if the ACK
bit is set in the TCP header; if it is valid, it echos a times-
tamp value that was sent by the remote TCP in the TSval field
of a Timestamps option. When TSecr is not valid, its value
must be zero. 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 Timestamps 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 re-
ceived a TSopt in the initial <SYN> segment for the connection.

3.3 The RTTM Mechanism

The timestamp value to be sent in TSval is to be obtained from a
(virtual) clock that we call the "timestamp clock". Its values
must be at least approximately proportional to real time, in order
to measure actual RTT.

The following example illustrates 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 Timestamps
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:

        A TSecr value received in a segment is used to update the
        averaged RTT measurement only if the segment acknowledges
        some new data, i.e., only if it advances the left edge of the
        send window.

   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.

3.4 Which Timestamp to Echo

   If more than one Timestamps 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 ACKs 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. On the other
     hand, an 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 Timestamps 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 ACKs have been delayed.

(2) If Last.ACK.sent falls within the range of sequence numbers
     of an incoming segment:

          SEG.SEQ <= Last.ACK.sent < SEG.SEQ + SEG.LEN

     then the TSval from the segment is copied to TS.Recent;
     otherwise, the TSval 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   Packets arrive in sequence, and some of the ACKs 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   Packets arrive out of order, and every packet 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: PROTECT 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 (Protect Against Wrapped Sequence numbers). PAWS
      operates within a single TCP connection, using state that is saved
      in the connection control block. Section 4.3 and Appendix C
      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 Timestamps 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 monotone 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 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 monotone 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 ACKs, 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 un-symmetric algorithm would protect against old
duplicate ACKs: 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.

It is recommended that RST segments NOT carry timestamps, and that
RST segments be acceptable regardless of their timestamp. Old
duplicate RST segments should be exceedingly unlikely, and their
cleanup function should take precedence over timestamps.

4.2.1 Basic PAWS Algorithm

    The PAWS algorithm requires the following processing to be
    performed on all incoming segments for a synchronized
    connection:

R1) If there is a Timestamps option in the arriving segment
     and SEG.TSval < TS.Recent and if TS.Recent is valid (see
     later discussion), then treat the arriving segment as not
     acceptable:

          Send an acknowledgement in reply as specified in
          RFC-793 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 RFC-793.

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 RFC-793.

It is important to note that the timestamp is 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.TStamp 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.Stamp 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 packet 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-R4
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 ACKs, then the
queued segments that caused these ACKs must have been received
already.

Even if a segment were delayed past the RTO, the Fast
Retransmit mechanism [Jacobson90c] will cause the delayed
packets 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.

4.2.2 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 RTT's. However,
the receiver treats the timestamp as simply a monotone-
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 RFC-1072 window
     extension, 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 1ms timestamp clock will work at
     link speeds up to 8 Tbps (8*10**12) bps!

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

     Its recycling time 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 4.2.3), 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 by the gateway
     system without requiring precise MSL enforcement by the
     TTL value in the IP layer.

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.

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

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

       *   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).

   4.2.3 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 keepalive
      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 keepalives, 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.

   4.2.4 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 win-
     dow and possibly sending acknowledgments, and queueing
     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 interval
arithmetic on a finite field, a relatively expensive operation.
To perform this check on every single segment is contrary to
the philosophy of header prediction. We believe that this
change might reduce CPU time for TCP protocol processing by up
to 5-10% 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 [Jacobson90b] 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 an FDDI 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.

4.3. Duplicates from Earlier Incarnations of Connection

The PAWS mechanism protects against errors due to sequence number
wrap-around on high-speed connection. 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 ACKNOWLEDGMENTS

   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's that do not implement
   the extensions.

   These mechanisms are implemented using new TCP options for scaled
   windows and timestamps. The timestamps are used for two distinct
   mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protect
   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 RFC-1185.

   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. More recently,
   task force members and other on the end2end-interest list have made
   valuable contributions by pointing out flaws in the algorithms and
   the documentation. The authors are grateful for all these
   contributions.

6. REFERENCES

   [Clark87] Clark, D., Lambert, M., and L. Zhang, "NETBLT: A Bulk
   Data Transfer Protocol", RFC 998, MIT, March 1987.

   [Garlick77] Garlick, L., R. Rom, and J. Postel, "Issues in
   Reliable Host-to-Host Protocols", Proc. Second Berkeley Workshop
   on Distributed Data Management and Computer Networks, May 1977.

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

   [Cheriton88] Cheriton, D., "VMTP: Versatile Message Transaction
   Protocol", RFC 1045, Stanford University, February 1988.

   [Jacobson88a] Jacobson, V., "Congestion Avoidance and Control",
   SIGCOMM '88, Stanford, CA., August 1988.

   [Jacobson88b] Jacobson, V., and R. Braden, "TCP Extensions for
   Long-Delay Paths", RFC-1072, LBL and USC/Information Sciences
   Institute, October 1988.

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

[Jacobson90b] Jacobson, V., Braden, R., and Zhang, L., "TCP
Extension for High-Speed Paths", RFC-1185, LBL and USC/Information
Sciences Institute, October 1990.

[Jacobson90c] Jacobson, V., "Modified TCP congestion avoidance
algorithm", Message to end2end-interest mailing list, April 1990.

[Jain86] Jain, R., "Divergence of Timeout Algorithms for Packet
Retransmissions", Proc. Fifth Phoenix Conf. on Comp. and Comm.,
Scottsdale, Arizona, March 1986.

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

[McKenzie89] McKenzie, A., "A Problem with the TCP Big Window
Option", RFC 1110, BBN STC, August 1989.

[Nagle84] Nagle, J., "Congestion Control in IP/TCP
Internetworks", RFC 896, FACC, January 1984.

[NBS85] Colella, R., Aronoff, R., and K. Mills, "Performance
Improvements for ISO Transport", Ninth Data Comm Symposium,
published in ACM SIGCOMM Comp Comm Review, vol. 15, no. 5,
September 1985.

[Postel81] Postel, J., "Transmission Control Protocol - DARPA
Internet Program Protocol Specification", RFC 793, DARPA,
September 1981.

[Velten84] Velten, D., Hinden, R., and J. Sax, "Reliable Data
Protocol", RFC 908, BBN, July 1984.

[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

   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        |
        +--------+--------+--------+--------+
```


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 [Postel81]. The MSL that is required here
      does not depend upon the transfer speed. The current TCP MSL of 2
      minutes seems acceptable as an operational compromise, as many
      host systems take this long to boot after a crash.

      However, 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 will be
      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
[Postel81]. 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 is
     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: CHANGES FROM RFC-1072, RFC-1185

   The protocol extensions defined in this document differ in several
   important ways from those defined in RFC-1072 and RFC-1185.

   (a)   SACK has been deferred to a later memo.

   (b)   The detailed rules for sending timestamp replies (see Section
         3.4) differ in important ways. The earlier rules could result
         in an under-estimate of the RTT in certain cases (packets
         dropped or out of order).

   (c)   The same value TS.Recent is now shared by the two distinct
         mechanisms RTTM and PAWS. This simplification became possible
         because of change (b).

   (d)   An ambiguity in RFC-1185 was resolved in favor of putting
         timestamps on ACK as well as data segments. This supports the
         symmetry of the underlying TCP protocol.

   (e)   The echo and echo reply options of RFC-1072 were combined into a
         single Timestamps option, to reflect the symmetry and to
         simplify processing.

   (f)   The problem of outdated timestamps on long-idle connections,
         discussed in Section 4.2.2, was realized and resolved.

   (g)   RFC-1185 recommended that header prediction take precedence over
         the timestamp check. Based upon some scepticism about the
         probabilistic arguments given in Section 4.2.4, it was decided
         to recommend that the timestamp check be performed first.

   (h)   The spec was modified so that the extended options will be sent
         on <SYN,ACK> segments only when they are received in the
         corresponding <SYN> segments. This provides the most
         conservative possible conditions for interoperation with
         implementations without the extensions.

   In addition to these substantive changes, the present RFC attempts to
   specify the algorithms unambiguously by presenting modifications to
   the Event Processing rules of RFC-793; see Appendix E.

APPENDIX D: SUMMARY OF NOTATION

    The following notation has been used in this document.

    Options

        WSopt: TCP Window Scale Option
        TSopt: TCP Timestamps 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: Local source of 32-bit timestamp values
        my.TSclock.rate: Period of my.TSclock (1 ms to 1 sec).

    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

APPENDIX E: EVENT PROCESSING

Event Processing

  OPEN Call

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

          <SEQ=ISS><CTL=SYN><TSval=my.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=my.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 Timestamps
       option <TSval=my.TSclock,TSecr=TS.Recent> in each data segment.

       Scale the receive window for transmission in the segment header:

            SEG.WND = (SND.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=my.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 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 my.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=my.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=my.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 Timestamps option and bit
Snd.TS.OK is on. If so:

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

If SEG.SEQ is equal to Last.ACK.sent, then save SEG.ECopt 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 Timestamps option
<TSval=my.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 my.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 Timestamps option
<TSval=my.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.

    ...

Security Considerations

    Security issues are not discussed in this memo.

Authors' Addresses

    Van Jacobson
    University of California
    Lawrence Berkeley Laboratory
    Mail Stop 46A
    Berkeley, CA 94720

    Phone: (415) 486-6411
    EMail: van@CSAM.LBL.GOV

    Bob Braden
    University of Southern California
    Information Sciences Institute
    4676 Admiralty Way
    Marina del Rey, CA 90292

    Phone: (310) 822-1511
    EMail: Braden@ISI.EDU

    Dave Borman
    Cray Research
    655-E Lone Oak Drive
    Eagan, MN 55121

    Phone: (612) 683-5571
    Email: dab@cray.com