TCP Extensions for Long-Delay Paths

Status of This Memo

This memo proposes a set of extensions to the TCP protocol to provide efficient operation over a path with a high bandwidth*delay product. These extensions are not proposed as an Internet standard at this time. Instead, they are intended as a basis for further experimentation and research on transport protocol performance. Distribution of this memo is unlimited.

1. INTRODUCTION

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 [Jacobson88]. However, there is still a fundamental TCP performance bottleneck for one transmission regime: paths with high bandwidth and long round-trip delays. The significant parameter is the product of bandwidth (bits per second) and round-trip delay (RTT in seconds); this product is the number of bits it takes to "fill the pipe", i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when this product is large, e.g., significantly exceeds $10^{**}5$ bits. We will 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 T1-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! Proposed future 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.

Clever algorithms alone will not give us good TCP performance over LFN's; it will be necessary to actually extend the protocol. This RFC proposes a set of TCP extensions for this purpose.

There are three fundamental problems with the current TCP over LFN

paths:

(1) Window Size Limitation

     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. (In practice, some TCP
     implementations will "break" for windows exceeding 2**15,
     because of their failure to do unsigned arithmetic).

     To circumvent this problem, we propose a new TCP option to allow
     windows larger than 2**16. This option will define an implicit
     scale factor, to be used to multiply the window size value found
     in a TCP header to obtain the true window size.

(2) Cumulative Acknowledgments

     Any packet losses in an LFN can have a catastrophic effect on
     throughput. This effect is exaggerated by the simple cumulative
     acknowledgment of TCP. Whenever a segment is lost, the
     transmitting TCP will (eventually) time out and retransmit the
     missing segment. However, the sending TCP has no information
     about segments that may have reached the receiver and been
     queued because they were not at the left window edge, so it may
     be forced to retransmit these segments unnecessarily.

     We propose a TCP extension to implement selective
     acknowledgements. By sending selective acknowledgments, the
     receiver of data can inform the sender about all segments that
     have arrived successfully, so the sender need retransmit only
     the segments that have actually been lost.

     Selective acknowledgments have been included in a number of
     experimental Internet protocols -- VMTP [Cheriton88], NETBLT
     [Clark87], and RDP [Velten84]. There is some empirical evidence
     in favor of selective acknowledgments -- simple experiments with
     RDP have shown that disabling the selective acknowlegment
     facility greatly increases the number of retransmitted segments
     over a lossy, high-delay Internet path [Partridge87]. A
     simulation study of a simple form of selective acknowledgments
     added to the ISO transport protocol TP4 also showed promise of
     performance improvement [NBS85].

(3) Round Trip Timing

TCP implements reliable data delivery by measuring the RTT, i.e., the time interval between sending a segment and receiving an acknowledgment for it, and retransmitting any segments that are not acknowledged within some small multiple of the average RTT. Experience has shown that accurate, current RTT estimates are necessary to adapt to changing traffic conditions and, without them, a busy network is subject to an instability known as "congestion collapse" [Nagle84].

In part because TCP segments may be repacketized upon retransmission, and in part because of complications due to the cumulative TCP acknowledgement, measuring a segments's RTT may involve a non-trivial amount of computation in some implementations. To minimize this computation, some implementations time only one segment per window. While this yields an adequate approximation to the RTT for small windows (e.g., a 4 to 8 segment Arpanet window), for an LFN (e.g., 100 segment Wideband Network windows) it results in an unacceptably poor RTT estimate.

In the presence of errors, 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 Net), it becomes effectively impossible to obtain an RTT measurement.

We propose a TCP "echo" option that allows each segment to carry its own timestamp. This will allow every segment, including retransmissions, to be timed at negligible computational cost.

In designing new TCP options, we must pay careful attention to interoperability with existing implementations. The only TCP option defined to date is an "initial option", i.e., it may appear only on a SYN segment. It is likely that most implementations will properly ignore any options in the SYN segment that they do not understand, so new initial options should not cause a problem. On the other hand, we fear that receiving unexpected non-initial options may cause some TCP's to crash.

Therefore, in each of the extensions we propose, non-initial options
may be sent only if an exchange of initial options has indicated that
both sides understand the extension. This approach will also allow a
TCP to determine when the connection opens how big a TCP header it
will be sending.

2. TCP WINDOW SCALE OPTION

The obvious way to implement a window scale factor would be to define
a new TCP option that could be included in any segment specifying a
window. The receiver would include it in every acknowledgment
segment, and the sender would interpret it. Unfortunately, this
simple approach would not work. The sender must reliably know the
receiver's current scale factor, but a TCP option in an
acknowledgement segment will not be delivered reliably (unless the
ACK happens to be piggy-backed on data).

However, SYN segments are always sent reliably, suggesting that each
side may communicate its window scale factor in an initial TCP
option. This approach has a disadvantage: the scale must be
established when the connection is opened, and cannot be changed
thereafter. However, other alternatives would be much more
complicated, and we therefore propose a new initial option called
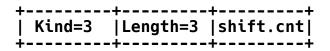Window Scale.

2.1 Window Scale Option

This three-byte option may be sent in a SYN segment by a TCP (1)
to indicate that it is prepared to do both send and receive window
scaling, and (2) to communicate a scale factor to be applied to
its receive window. The scale factor is encoded logarithmically,
as a power of 2 (presumably to be implemented by binary shifts).

Note: the window in the SYN segment itself is never scaled.

TCP Window Scale Option:

Kind: 3

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

Here shift.cnt is the number of bits by which the receiver right-
shifts the true receive-window value, to scale it into a 16-bit
value to be sent in TCP header (this scaling is explained below).
The value shift.cnt may be zero (offering to scale, while applying
a scale factor of 1 to the receive window).

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.

2.2 Using the Window Scale Option

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

*   The send-window (SND.WND) and receive-window (RCV.WND) sizes
        in the connection state block and in all sequence space
        calculations are expanded from 16 to 32 bits.

*   Two window shift counts are added to the connection state:
        snd.scale and rcv.scale. These are shift counts to be
        applied to the incoming and outgoing windows, respectively.
        The precise algorithm is shown below.

*   All outgoing SYN segments are sent with the Window Scale
        option, containing a value shift.cnt = R that the TCP would
        like to use for its receive window.

*   Snd.scale and rcv.scale are initialized to zero, and are
        changed only during processing of a received SYN segment. If
        the SYN segment contains a Window Scale option with shift.cnt
        = S, set snd.scale to S and set rcv.scale to R; otherwise,
        both snd.scale and rcv.scale are left at zero.

*   The window field (SEG.WND) in the header of every incoming
        segment, with the exception of SYN segments, will be left-
        shifted by snd.scale bits before updating SND.WND:

            SND.WND = SEG.WND << snd.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, will have been right-shifted
        by rcv.scale bits:

            SEG.WND = RCV.WND >> rcv.scale.

TCP determines if a data segment is "old" or "new" by testing if
its sequence number is within 2^{31} bytes of the left edge of the
window. If not, the data is "old" and discarded. 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 least 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. (This 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.

3. TCP SELECTIVE ACKNOWLEDGMENT OPTIONS

To minimize the impact on the TCP protocol, the selective
acknowledgment extension uses the form of two new TCP options. The
first is an enabling option, "SACK-permitted", that may be sent in a
SYN segment to indicate that the the SACK option may be used once the
connection is established. The other is the SACK option itself,
which may be sent over an established connection once permission has
been given by SACK-permitted.

The SACK option is to be included in a segment sent from a TCP that
is receiving data to the TCP that is sending that data; we will refer
to these TCP's as the data receiver and the data sender,
respectively. We will consider a particular simplex data flow; any
data flowing in the reverse direction over the same connection can be
treated independently.

3.1 SACK-Permitted Option

This two-byte option may be sent in a SYN by a TCP that has been
extended to receive (and presumably process) the SACK option once
the connection has opened.

TCP Sack-Permitted Option:

Kind: 4

```
        +---------+---------+
        | Kind=4  | Length=2|
        +---------+---------+
```

3.2 SACK Option

The SACK option is to be used to convey extended acknowledgment
information over an established connection. Specifically, it is
to be sent by a data receiver to inform the data transmitter of
non-contiguous blocks of data that have been received and queued.
The data receiver is awaiting the receipt of data in later
retransmissions to fill the gaps in sequence space between these
blocks. At that time, the data receiver will acknowledge the data
normally by advancing the left window edge in the Acknowledgment
Number field of the TCP header.

It is important to understand that the SACK option will not change
the meaning of the Acknowledgment Number field, whose value will
still specify the left window edge, i.e., one byte beyond the last
sequence number of fully-received data. The SACK option is
advisory; if it is ignored, TCP acknowledgments will continue to
function as specified in the protocol.

However, SACK will provide additional information that the data
transmitter can use to optimize retransmissions. The TCP data
receiver may include the SACK option in an acknowledgment segment
whenever it has data that is queued and unacknowledged. Of
course, the SACK option may be sent only when the TCP has received
the SACK-permitted option in the SYN segment for that connection.

TCP SACK Option:

Kind: 5

Length: Variable

```
  +--------+--------+--------+--------+--------+--------+...--+
  | Kind=5 | Length | Relative Origin |   Block Size    |    |
  +--------+--------+--------+--------+--------+--------+...--+
```

This option contains a list of the blocks of contiguous sequence
space occupied by data that has been received and queued within

the window. Each block is contiguous and isolated; that is, the octets just below the block,

Acknowledgment Number + Relative Origin -1,

and just above the block,

Acknowledgment Number + Relative Origin + Block Size,

have not been received.

Each contiguous block of data queued at the receiver is defined in the SACK option by two 16-bit integers:

*   Relative Origin

This is the first sequence number of this block, relative to the Acknowledgment Number field in the TCP header (i.e., relative to the data receiver's left window edge).

*   Block Size

This is the size in octets of this block of contiguous data.

A SACK option that specifies n blocks will have a length of 4*n+2 octets, so the 44 bytes available for TCP options can specify a maximum of 10 blocks. Of course, if other TCP options are introduced, they will compete for the 44 bytes, and the limit of 10 may be reduced in particular segments.

There is no requirement on the order in which blocks can appear in a single SACK option.

Note: requiring that the blocks be ordered would allow a slightly more efficient algorithm in the transmitter; however, this does not seem to be an important optimization.

3.3 SACK with Window Scaling

If window scaling is in effect, then 16 bits may not be sufficient for the SACK option fields that define the origin and length of a block. There are two possible ways to handle this:

(1) Expand the SACK origin and length fields to 24 or 32 bits.

(2) Scale the SACK fields by the same factor as the window.

The first alternative would significantly reduce the number of
blocks possible in a SACK option; therefore, we have chosen the
second alternative, scaling the SACK information as well as the
window.

Scaling the SACK information introduces some loss of precision,
since a SACK option must report queued data blocks whose origins
and lengths are multiples of the window scale factor rcv.scale.
These reported blocks must be equal to or smaller than the actual
blocks of queued data.

Specifically, suppose that the receiver has a contiguous block of
queued data that occupies sequence numbers L, L+1, ... L+N-1, and
that the window scale factor is S = rcv.scale. Then the
corresponding block that will be reported in a SACK option will
be:

    Relative Origin = int((L+S-1)/S)

    Block Size = int((L+N)/S) - (Relative Origin)

where the function int(x) returns the greatest integer contained
in x.

The resulting loss of precision is not a serious problem for the
sender. If the data-sending TCP keeps track of the boundaries of
all segments in its retransmission queue, it will generally be
able to infer from the imprecise SACK data which full segments
don't need to be retransmitted. This will fail only if S is
larger than the maximum segment size, in which case some segments
may be retransmitted unnecessarily. If the sending TCP does not
keep track of transmitted segment boundaries, the imprecision of
the scaled SACK quantities will only result in retransmitting a
small amount of unneeded sequence space. On the average, the data
sender will unnecessarily retransmit J*S bytes of the sequence
space for each SACK received; here J is the number of blocks
reported in the SACK, and S = snd.scale.

3.4 SACK Option Examples

Assume the left window edge is 5000 and that the data transmitter
sends a burst of 8 segments, each containing 500 data bytes.
Unless specified otherwise, we assume that the scale factor S = 1.

Case 1: The first 4 segments are received but the last 4 are dropped.

The data receiver will return a normal TCP ACK segment acknowledging sequence number 7000, with no SACK option.

Case 2: The first segment is dropped but the remaining 7 are received.

The data receiver will return a TCP ACK segment that acknowledges sequence number 5000 and contains a SACK option specifying one block of queued data:

>   Relative Origin = 500; Block Size = 3500

Case 3: The 2nd, 4th, 6th, and 8th (last) segments are dropped.

The data receiver will return a TCP ACK segment that acknowledges sequence number 5500 and contains a SACK option specifying the 3 blocks:

>   Relative Origin = 500; Block Size = 500
>   Relative Origin = 1500; Block Size = 500
>   Relative Origin = 2500; Block Size = 500

Case 4: Same as Case 3, except Scale Factor S = 16.

The SACK option would specify the 3 scaled blocks:

>   Relative Origin =   32; Block Size = 30
>   Relative Origin =   94; Block Size = 31
>   Relative Origin = 157; Block Size = 30

These three reported blocks have sequence numbers 512 through 991, 1504 through 1999, and 2512 through 2992, respectively.

3.5 Generating the SACK Option

Let us assume that the data receiver maintains a queue of valid segments that it has neither passed to the user nor acknowledged because of earlier missing data, and that this queue is ordered by starting sequence number. Computation of the SACK option can be done with one pass down this queue. Segments that occupy

contiguous sequence space are aggregated into a single SACK block,
and each gap in the sequence space (except a gap that is
terminated by the right window edge) triggers the start of a new
SACK block. If this algorithm defines more than 10 blocks, only
the first 10 can be included in the option.

3.6 Interpreting the SACK Option

The data transmitter is assumed to have a retransmission queue
that contains the segments that have been transmitted but not yet
acknowledged, in sequence-number order. If the data transmitter
performs re-packetization before retransmission, the block
boundaries in a SACK option that it receives may not fall on
boundaries of segments in the retransmission queue; however, this
does not pose a serious difficulty for the transmitter.

Let us suppose that for each segment in the retransmission queue
there is a (new) flag bit "ACK'd", to be used to indicate that
this particular segment has been entirely acknowledged. When a
segment is first transmitted, it will be entered into the
retransmission queue with its ACK'd bit off. If the ACK'd bit is
subsequently turned on (as the result of processing a received
SACK option), the data transmitter will skip this segment during
any later retransmission. However, the segment will not be
dequeued and its buffer freed until the left window edge is
advanced over it.

When an acknowledgment segment arrives containing a SACK option,
the data transmitter will turn on the ACK'd bits for segments that
have been selectively acknowleged. More specifically, for each
block in the SACK option, the data transmitter will turn on the
ACK'd flags for all segments in the retransmission queue that are
wholly contained within that block. This requires straightforward
sequence number comparisons.

4. TCP ECHO OPTIONS

A simple method for measuring the RTT of a segment would be: the
sender places a timestamp in the segment and the receiver returns
that timestamp in the corresponding ACK segment. When the ACK segment
arrives at the sender, the difference between the current time and
the timestamp is the RTT. To implement this timing method, the
receiver must simply reflect or echo selected data (the timestamp)
from the sender's segments. This idea is the basis of the "TCP Echo"
and "TCP Echo Reply" options.

4.1   TCP Echo and TCP Echo Reply Options

     TCP Echo Option:

     Kind: 6

     Length: 6

```
     +--------+--------+--------+--------+--------+--------+
     | Kind=6 | Length |   4 bytes of info to be echoed   |
     +--------+--------+--------+--------+--------+--------+
```

   This option carries four bytes of information that the receiving TCP
   may send back in a subsequent TCP Echo Reply option (see below). A
   TCP may send the TCP Echo option in any segment, but only if a TCP
   Echo option was received in a SYN segment for the connection.

   When the TCP echo option is used for RTT measurement, it will be
   included in data segments, and the four information bytes will define
   the time at which the data segment was transmitted in any format
   convenient to the sender.

   TCP Echo Reply Option:

   Kind: 7

   Length: 6

```
     +--------+--------+--------+--------+--------+--------+
     | Kind=7 | Length |    4 bytes of echoed info        |
     +--------+--------+--------+--------+--------+--------+
```

   A TCP that receives a TCP Echo option containing four information
   bytes will return these same bytes in a TCP Echo Reply option.

   This TCP Echo Reply option must be returned in the next segment
   (e.g., an ACK segment) that is sent. If more than one Echo option is
   received before a reply segment is sent, the TCP must choose only one
   of the options to echo, ignoring the others; specifically, it must
   choose the newest segment with the oldest sequence number (see next
   section.)

   To use the TCP Echo and Echo Reply options, a TCP must send a TCP
   Echo option in its own SYN segment and receive a TCP Echo option in a
   SYN segment from the other TCP. A TCP that does not implement the
   TCP Echo or Echo Reply options must simply ignore any TCP Echo
   options it receives. However, a TCP should not receive one of these

options in a non-SYN segment unless it included a TCP Echo option in
its own SYN segment.

4.2 Using the Echo Options

If we wish to use the Echo/Echo Reply options for RTT measurement, we
have to define what the receiver does when there is not a one-to-one
correspondence between data and ACK segments. Assuming that we want
to minimize the state kept in the receiver (i.e., the number of
unprocessed Echo options), we can plan on a receiver remembering the
information value from at most one Echo between ACKs. 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 ACK's". The data-sender TCP must
     measure the effective RTT, including the additional time due to
     delayed ACK's, or else it will retransmit unnecessarily. Thus,
     when delayed ACK's are in use, the receiver should reply with
     the Echo option information 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
     we may be generating ACKs as these out-of-order segments arrive
     (e.g., for the SACK information or to aid "fast retransmit").
     An Echo Reply option will tell the sender the RTT of some
     recently sent segment (since the ACK can only contain the
     sequence number of the hole, the sender may not be able to
     determine which segment, but that doesn't matter). If the loss
     was due to congestion, these RTTs may be particularly valuable
     to the sender since they reflect the network characteristics
     immediately after the congestion.

(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.

Case (A) suggests the receiver should remember and return the Echo
option information from the oldest unacknowledged segment. Cases (B)

and (C) suggest that the option should come from the most recent
unacknowledged segment. An algorithm that covers all three cases is
for the receiver to return the Echo option information from the
newest segment with the oldest sequence number, as specified earlier.

A model implementation of these options is as follows.

(1) Receiver Implementation

     A 32-bit slot for Echo option data, rcv.echodata, is added to
     the receiver connection state, together with a flag,
     rcv.echopresent, that indicates whether there is anything in the
     slot. When the receiver generates a segment, it checks
     rcv.echopresent and, if it is set, adds an echo-reply option
     containing rcv.echodata to the outgoing segment then clears
     rcv.echopresent.

     If an incoming segment is in the window and contains an echo
     option, the receiver checks rcv.echopresent. If it isn't set,
     the value of the echo option is copied to rcv.echodata and
     rcv.echopresent is set. If rcv.echopresent is already set, the
     receiver checks whether the segment is at the left edge of the
     window. If so, the segment's echo option value is copied to
     rcv.echodata (this is situation (C) above). Otherwise, the
     segment's echo option is ignored.

(2) Sender Implementation

     The sender's connection state has a single flag bit,
     snd.echoallowed, added. If snd.echoallowed is set or if the
     segment contains a SYN, the sender is free to add a TCP Echo
     option (presumably containing the current time in some units
     convenient to the sender) to every outgoing segment.

     Snd.echoallowed should be set if a SYN is received with a TCP
     Echo option (presumably, a host that implements the option will
     attempt to use it to time the SYN segment).

5. CONCLUSIONS AND ACKNOWLEDGMENTS

We have proposed five new TCP options for scaled windows, selective
acknowledgments, and round-trip timing, in order to provide efficient
operation over large-bandwidth*delay-product paths. These extensions
are designed to provide compatible interworking with TCP's that do not
implement the extensions.

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 proposed by Van
Jacobson. Gerd Beling of FGAN (West Germany) contributed the initial
definition of the SACK option.

All three options have evolved through discussion with the End-to-End
Task Force, and the authors are grateful to the other members of the
Task Force for their advice and encouragement.

6. REFERENCES

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

[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.

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

[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.

[Partridge87] Partridge, C., "Private Communication", February
1987.

[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.

[Jacobson88] Jacobson, V., "Congestion Avoidance and Control", to
be presented at SIGCOMM '88, Stanford, CA., August 1988.

[Zhang86] Zhang, L., "Why TCP Timers Don't Work Well", Proc.

SIGCOMM '86, Stowe, Vt., August 1986.