                        TCP Congestion Control

Status of this Memo

Copyright Notice

Abstract

   This document defines TCP's four intertwined congestion control
   algorithms: slow start, congestion avoidance, fast retransmit, and
   fast recovery. In addition, the document specifies how TCP should
   begin transmission after a relatively long idle period, as well as
   discussing various acknowledgment generation methods.

1. Introduction

   This document specifies four TCP [Pos81] congestion control
   algorithms: slow start, congestion avoidance, fast retransmit and
   fast recovery. These algorithms were devised in [Jac88] and [Jac90].
   Their use with TCP is standardized in [Bra89].

   This document is an update of [Ste97]. In addition to specifying the
   congestion control algorithms, this document specifies what TCP
   connections should do after a relatively long idle period, as well as
   specifying and clarifying some of the issues pertaining to TCP ACK
   generation.

   Note that [Ste94] provides examples of these algorithms in action and
   [WS95] provides an explanation of the source code for the BSD
   implementation of these algorithms.

This document is organized as follows. Section 2 provides various definitions which will be used throughout the document. Section 3 provides a specification of the congestion control algorithms. Section 4 outlines concerns related to the congestion control algorithms and finally, section 5 outlines security considerations.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [Bra97].

2. Definitions

This section provides the definition of several terms that will be used throughout the remainder of this document.

SEGMENT:
   A segment is ANY TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [MD90] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, 536 bytes [Bra89]. The size does not include the TCP/IP headers and options.

FULL-SIZED SEGMENT: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd) The most recently advertised receiver window.

CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

INITIAL WINDOW (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

   LOSS WINDOW (LW): The loss window is the size of the congestion
      window after a TCP sender detects loss using its retransmission
      timer.

   RESTART WINDOW (RW): The restart window is the size of the
      congestion window after a TCP restarts transmission after an idle
      period (if the slow start algorithm is used; see section 4.1 for
      more discussion).

   FLIGHT SIZE: The amount of data that has been sent but not yet
      acknowledged.

3. Congestion Control Algorithms

   This section defines the four congestion control algorithms: slow
   start, congestion avoidance, fast retransmit and fast recovery,
   developed in [Jac88] and [Jac90]. In some situations it may be
   beneficial for a TCP sender to be more conservative than the
   algorithms allow, however a TCP MUST NOT be more aggressive than the
   following algorithms allow (that is, MUST NOT send data when the
   value of cwnd computed by the following algorithms would not allow
   the data to be sent).

3.1 Slow Start and Congestion Avoidance

   The slow start and congestion avoidance algorithms MUST be used by a
   TCP sender to control the amount of outstanding data being injected
   into the network. To implement these algorithms, two variables are
   added to the TCP per-connection state. The congestion window (cwnd)
   is a sender-side limit on the amount of data the sender can transmit
   into the network before receiving an acknowledgment (ACK), while the
   receiver's advertised window (rwnd) is a receiver-side limit on the
   amount of outstanding data. The minimum of cwnd and rwnd governs
   data transmission.

   Another state variable, the slow start threshold (ssthresh), is used
   to determine whether the slow start or congestion avoidance algorithm
   is used to control data transmission, as discussed below.

   Beginning transmission into a network with unknown conditions
   requires TCP to slowly probe the network to determine the available
   capacity, in order to avoid congesting the network with an
   inappropriately large burst of data. The slow start algorithm is
   used for this purpose at the beginning of a transfer, or after
   repairing loss detected by the retransmission timer.

IW, the initial value of cwnd, MUST be less than or equal to 2*SMSS
bytes and MUST NOT be more than 2 segments.

We note that a non-standard, experimental TCP extension allows that a
TCP MAY use a larger initial window (IW), as defined in equation 1
[AFP98]:

   IW = min (4*SMSS, max (2*SMSS, 4380 bytes))   (1)

With this extension, a TCP sender MAY use a 3 or 4 segment initial
window, provided the combined size of the segments does not exceed
4380 bytes. We do NOT allow this change as part of the standard
defined by this document. However, we include discussion of (1) in
the remainder of this document as a guideline for those experimenting
with the change, rather than conforming to the present standards for
TCP congestion control.

The initial value of ssthresh MAY be arbitrarily high (for example,
some implementations use the size of the advertised window), but it
may be reduced in response to congestion. The slow start algorithm
is used when cwnd < ssthresh, while the congestion avoidance
algorithm is used when cwnd > ssthresh. When cwnd and ssthresh are
equal the sender may use either slow start or congestion avoidance.

During slow start, a TCP increments cwnd by at most SMSS bytes for
each ACK received that acknowledges new data. Slow start ends when
cwnd exceeds ssthresh (or, optionally, when it reaches it, as noted
above) or when congestion is observed.

During congestion avoidance, cwnd is incremented by 1 full-sized
segment per round-trip time (RTT). Congestion avoidance continues
until congestion is detected. One formula commonly used to update
cwnd during congestion avoidance is given in equation 2:

   cwnd += SMSS*SMSS/cwnd   (2)

This adjustment is executed on every incoming non-duplicate ACK.
Equation (2) provides an acceptable approximation to the underlying
principle of increasing cwnd by 1 full-sized segment per RTT. (Note
that for a connection in which the receiver acknowledges every data
segment, (2) proves slightly more aggressive than 1 segment per RTT,
and for a receiver acknowledging every-other packet, (2) is less
aggressive.)

Implementation Note: Since integer arithmetic is usually used in TCP
implementations, the formula given in equation 2 can fail to increase
cwnd when the congestion window is very large (larger than
SMSS*SMSS). If the above formula yields 0, the result SHOULD be
rounded up to 1 byte.

Implementation Note: older implementations have an additional
additive constant on the right-hand side of equation (2). This is
incorrect and can actually lead to diminished performance [PAD+98].

Another acceptable way to increase cwnd during congestion avoidance
is to count the number of bytes that have been acknowledged by ACKs
for new data. (A drawback of this implementation is that it requires
maintaining an additional state variable.) When the number of bytes
acknowledged reaches cwnd, then cwnd can be incremented by up to SMSS
bytes. Note that during congestion avoidance, cwnd MUST NOT be
increased by more than the larger of either 1 full-sized segment per
RTT, or the value computed using equation 2.

Implementation Note: some implementations maintain cwnd in units of
bytes, while others in units of full-sized segments. The latter will
find equation (2) difficult to use, and may prefer to use the
counting approach discussed in the previous paragraph.

When a TCP sender detects segment loss using the retransmission
timer, the value of ssthresh MUST be set to no more than the value
given in equation 3:

    ssthresh = max (FlightSize / 2, 2*SMSS)   (3)

As discussed above, FlightSize is the amount of outstanding data in
the network.

Implementation Note: an easy mistake to make is to simply use cwnd,
rather than FlightSize, which in some implementations may
incidentally increase well beyond rwnd.

Furthermore, upon a timeout cwnd MUST be set to no more than the loss
window, LW, which equals 1 full-sized segment (regardless of the
value of IW). Therefore, after retransmitting the dropped segment
the TCP sender uses the slow start algorithm to increase the window
from 1 full-sized segment to the new value of ssthresh, at which
point congestion avoidance again takes over.

3.2 Fast Retransmit/Fast Recovery

   A TCP receiver SHOULD send an immediate duplicate ACK when an out-
   of-order segment arrives. The purpose of this ACK is to inform the
   sender that a segment was received out-of-order and which sequence
   number is expected. From the sender's perspective, duplicate ACKs
   can be caused by a number of network problems. First, they can be
   caused by dropped segments. In this case, all segments after the
   dropped segment will trigger duplicate ACKs. Second, duplicate ACKs
   can be caused by the re-ordering of data segments by the network (not
   a rare event along some network paths [Pax97]). Finally, duplicate
   ACKs can be caused by replication of ACK or data segments by the
   network. In addition, a TCP receiver SHOULD send an immediate ACK
   when the incoming segment fills in all or part of a gap in the
   sequence space. This will generate more timely information for a
   sender recovering from a loss through a retransmission timeout, a
   fast retransmit, or an experimental loss recovery algorithm, such as
   NewReno [FH98].

   The TCP sender SHOULD use the "fast retransmit" algorithm to detect
   and repair loss, based on incoming duplicate ACKs. The fast
   retransmit algorithm uses the arrival of 3 duplicate ACKs (4
   identical ACKs without the arrival of any other intervening packets)
   as an indication that a segment has been lost. After receiving 3
   duplicate ACKs, TCP performs a retransmission of what appears to be
   the missing segment, without waiting for the retransmission timer to
   expire.

   After the fast retransmit algorithm sends what appears to be the
   missing segment, the "fast recovery" algorithm governs the
   transmission of new data until a non-duplicate ACK arrives. The
   reason for not performing slow start is that the receipt of the
   duplicate ACKs not only indicates that a segment has been lost, but
   also that segments are most likely leaving the network (although a
   massive segment duplication by the network can invalidate this
   conclusion). In other words, since the receiver can only generate a
   duplicate ACK when a segment has arrived, that segment has left the
   network and is in the receiver's buffer, so we know it is no longer
   consuming network resources. Furthermore, since the ACK "clock"
   [Jac88] is preserved, the TCP sender can continue to transmit new
   segments (although transmission must continue using a reduced cwnd).

   The fast retransmit and fast recovery algorithms are usually
   implemented together as follows.

   1. When the third duplicate ACK is received, set ssthresh to no more
       than the value given in equation 3.

   2. Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS.
       This artificially "inflates" the congestion window by the number
       of segments (three) that have left the network and which the
       receiver has buffered.

   3. For each additional duplicate ACK received, increment cwnd by
       SMSS. This artificially inflates the congestion window in order
       to reflect the additional segment that has left the network.

   4. Transmit a segment, if allowed by the new value of cwnd and the
       receiver's advertised window.

   5. When the next ACK arrives that acknowledges new data, set cwnd to
       ssthresh (the value set in step 1). This is termed "deflating"
       the window.

       This ACK should be the acknowledgment elicited by the
       retransmission from step 1, one RTT after the retransmission
       (though it may arrive sooner in the presence of significant out-
       of-order delivery of data segments at the receiver).
       Additionally, this ACK should acknowledge all the intermediate
       segments sent between the lost segment and the receipt of the
       third duplicate ACK, if none of these were lost.

   Note: This algorithm is known to generally not recover very
   efficiently from multiple losses in a single flight of packets
   [FF96]. One proposed set of modifications to address this problem
   can be found in [FH98].

4. Additional Considerations

4.1 Re-starting Idle Connections

   A known problem with the TCP congestion control algorithms described
   above is that they allow a potentially inappropriate burst of traffic
   to be transmitted after TCP has been idle for a relatively long
   period of time. After an idle period, TCP cannot use the ACK clock
   to strobe new segments into the network, as all the ACKs have drained
   from the network. Therefore, as specified above, TCP can potentially
   send a cwnd-size line-rate burst into the network after an idle
   period.

   [Jac88] recommends that a TCP use slow start to restart transmission
   after a relatively long idle period. Slow start serves to restart
   the ACK clock, just as it does at the beginning of a transfer. This
   mechanism has been widely deployed in the following manner. When TCP
   has not received a segment for more than one retransmission timeout,
   cwnd is reduced to the value of the restart window (RW) before

transmission begins.

For the purposes of this standard, we define RW = IW.

We note that the non-standard experimental extension to TCP defined
in [AFP98] defines RW = min(IW, cwnd), with the definition of IW
adjusted per equation (1) above.

Using the last time a segment was received to determine whether or
not to decrease cwnd fails to deflate cwnd in the common case of
persistent HTTP connections [HTH98]. In this case, a WWW server
receives a request before transmitting data to the WWW browser. The
reception of the request makes the test for an idle connection fail,
and allows the TCP to begin transmission with a possibly
inappropriately large cwnd.

Therefore, a TCP SHOULD set cwnd to no more than RW before beginning
transmission if the TCP has not sent data in an interval exceeding
the retransmission timeout.

4.2 Generating Acknowledgments

The delayed ACK algorithm specified in [Bra89] SHOULD be used by a
TCP receiver. When used, a TCP receiver MUST NOT excessively delay
acknowledgments. Specifically, an ACK SHOULD be generated for at
least every second full-sized segment, and MUST be generated within
500 ms of the arrival of the first unacknowledged packet.

The requirement that an ACK "SHOULD" be generated for at least every
second full-sized segment is listed in [Bra89] in one place as a
SHOULD and another as a MUST. Here we unambiguously state it is a
SHOULD. We also emphasize that this is a SHOULD, meaning that an
implementor should indeed only deviate from this requirement after
careful consideration of the implications. See the discussion of
"Stretch ACK violation" in [PAD+98] and the references therein for a
discussion of the possible performance problems with generating ACKs
less frequently than every second full-sized segment.

In some cases, the sender and receiver may not agree on what
constitutes a full-sized segment. An implementation is deemed to
comply with this requirement if it sends at least one acknowledgment
every time it receives 2*RMSS bytes of new data from the sender,
where RMSS is the Maximum Segment Size specified by the receiver to
the sender (or the default value of 536 bytes, per [Bra89], if the
receiver does not specify an MSS option during connection
establishment). The sender may be forced to use a segment size less
than RMSS due to the maximum transmission unit (MTU), the path MTU
discovery algorithm or other factors. For instance, consider the

case when the receiver announces an RMSS of X bytes but the sender
ends up using a segment size of Y bytes (Y < X) due to path MTU
discovery (or the sender's MTU size). The receiver will generate
stretch ACKs if it waits for 2*X bytes to arrive before an ACK is
sent. Clearly this will take more than 2 segments of size Y bytes.
Therefore, while a specific algorithm is not defined, it is desirable
for receivers to attempt to prevent this situation, for example by
acknowledging at least every second segment, regardless of size.
Finally, we repeat that an ACK MUST NOT be delayed for more than 500
ms waiting on a second full-sized segment to arrive.

Out-of-order data segments SHOULD be acknowledged immediately, in
order to accelerate loss recovery. To trigger the fast retransmit
algorithm, the receiver SHOULD send an immediate duplicate ACK when
it receives a data segment above a gap in the sequence space. To
provide feedback to senders recovering from losses, the receiver
SHOULD send an immediate ACK when it receives a data segment that
fills in all or part of a gap in the sequence space.

A TCP receiver MUST NOT generate more than one ACK for every incoming
segment, other than to update the offered window as the receiving
application consumes new data [page 42, Pos81][Cla82].

4.3 Loss Recovery Mechanisms

A number of loss recovery algorithms that augment fast retransmit and
fast recovery have been suggested by TCP researchers. While some of
these algorithms are based on the TCP selective acknowledgment (SACK)
option [MMFR96], such as [FF96,MM96a,MM96b], others do not require
SACKs [Hoe96,FF96,FH98]. The non-SACK algorithms use "partial
acknowledgments" (ACKs which cover new data, but not all the data
outstanding when loss was detected) to trigger retransmissions.
While this document does not standardize any of the specific
algorithms that may improve fast retransmit/fast recovery, these
enhanced algorithms are implicitly allowed, as long as they follow
the general principles of the basic four algorithms outlined above.

Therefore, when the first loss in a window of data is detected,
ssthresh MUST be set to no more than the value given by equation (3).
Second, until all lost segments in the window of data in question are
repaired, the number of segments transmitted in each RTT MUST be no
more than half the number of outstanding segments when the loss was
detected. Finally, after all loss in the given window of segments
has been successfully retransmitted, cwnd MUST be set to no more than
ssthresh and congestion avoidance MUST be used to further increase
cwnd. Loss in two successive windows of data, or the loss of a
retransmission, should be taken as two indications of congestion and,
therefore, cwnd (and ssthresh) MUST be lowered twice in this case.

The algorithms outlined in [Hoe96,FF96,MM96a,MM6b] follow the
principles of the basic four congestion control algorithms outlined
in this document.

5. Security Considerations

This document requires a TCP to diminish its sending rate in the
presence of retransmission timeouts and the arrival of duplicate
acknowledgments. An attacker can therefore impair the performance of
a TCP connection by either causing data packets or their
acknowledgments to be lost, or by forging excessive duplicate
acknowledgments. Causing two congestion control events back-to-back
will often cut ssthresh to its minimum value of 2*SMSS, causing the
connection to immediately enter the slower-performing congestion
avoidance phase.

The Internet to a considerable degree relies on the correct
implementation of these algorithms in order to preserve network
stability and avoid congestion collapse. An attacker could cause TCP
endpoints to respond more aggressively in the face of congestion by
forging excessive duplicate acknowledgments or excessive
acknowledgments for new data. Conceivably, such an attack could
drive a portion of the network into congestion collapse.

6. Changes Relative to RFC 2001

This document has been extensively rewritten editorially and it is
not feasible to itemize the list of changes between the two
documents. The intention of this document is not to change any of the
recommendations given in RFC 2001, but to further clarify cases that
were not discussed in detail in 2001. Specifically, this document
suggests what TCP connections should do after a relatively long idle
period, as well as specifying and clarifying some of the issues
pertaining to TCP ACK generation. Finally, the allowable upper bound
for the initial congestion window has also been raised from one to
two segments.

Acknowledgments

The four algorithms that are described were developed by Van
Jacobson.

Some of the text from this document is taken from "TCP/IP
Illustrated, Volume 1: The Protocols" by W. Richard Stevens
(Addison-Wesley, 1994) and "TCP/IP Illustrated, Volume 2: The
Implementation" by Gary R. Wright and W. Richard Stevens (Addison-
Wesley, 1995). This material is used with the permission of
Addison-Wesley.

   Neal Cardwell, Sally Floyd, Craig Partridge and Joe Touch contributed
   a number of helpful suggestions.

References

   [AFP98]   Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's
             Initial Window Size, RFC 2414, September 1998.

   [Bra89]   Braden, R., "Requirements for Internet Hosts --
             Communication Layers", STD 3, RFC 1122, October 1989.

   [Bra97]   Bradner, S., "Key words for use in RFCs to Indicate
             Requirement Levels", BCP 14, RFC 2119, March 1997.

   [Cla82]   Clark, D., "Window and Acknowledgment Strategy in TCP", RFC
             813, July 1982.

   [FF96]   Fall, K. and S. Floyd, "Simulation-based Comparisons of
             Tahoe, Reno and SACK TCP", Computer Communication Review,
             July 1996. ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z.

   [FH98]   Floyd, S. and T. Henderson, "The NewReno Modification to
             TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

   [Flo94]   Floyd, S., "TCP and Successive Fast Retransmits. Technical
             report", October 1994.
             ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

   [Hoe96]   Hoe, J., "Improving the Start-up Behavior of a Congestion
             Control Scheme for TCP", In ACM SIGCOMM, August 1996.

   [HTH98]   Hughes, A., Touch, J. and J. Heidemann, "Issues in TCP
             Slow-Start Restart After Idle", Work in Progress.

   [Jac88]   Jacobson, V., "Congestion Avoidance and Control", Computer
             Communication Review, vol. 18, no. 4, pp. 314-329, Aug.
             1988. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

   [Jac90]   Jacobson, V., "Modified TCP Congestion Avoidance Algorithm",
             end2end-interest mailing list, April 30, 1990.
             ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail.

   [MD90]   Mogul, J. and S. Deering, "Path MTU Discovery", RFC 1191,
             November 1990.

   [MM96a]   Mathis, M. and J. Mahdavi, "Forward Acknowledgment: Refining
              TCP Congestion Control", Proceedings of SIGCOMM'96, August,
              1996, Stanford, CA. Available
              fromhttp://www.psc.edu/networking/papers/papers.html

   [MM96b]   Mathis, M. and J. Mahdavi, "TCP Rate-Halving with Bounding
              Parameters", Technical report. Available from
              http://www.psc.edu/networking/papers/FACKnotes/current.

   [MMFR96] Mathis, M., Mahdavi, J., Floyd, S. and A. Romanow, "TCP
              Selective Acknowledgement Options", RFC 2018, October 1996.

   [PAD+98] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J.,
              Heavens, I., Lahey, K., Semke, J. and B. Volz, "Known TCP
              Implementation Problems", RFC 2525, March 1999.

   [Pax97]   Paxson, V., "End-to-End Internet Packet Dynamics",
              Proceedings of SIGCOMM '97, Cannes, France, Sep. 1997.

   [Pos81]   Postel, J., "Transmission Control Protocol", STD 7, RFC 793,
              September 1981.

   [Ste94]   Stevens, W., "TCP/IP Illustrated, Volume 1: The Protocols",
              Addison-Wesley, 1994.

   [Ste97]   Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast
              Retransmit, and Fast Recovery Algorithms", RFC 2001, January
              1997.

   [WS95]   Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2:
              The Implementation", Addison-Wesley, 1995.

Authors' Addresses

   Mark Allman
   NASA Glenn Research Center/Sterling Software
   Lewis Field
   21000 Brookpark Rd.   MS 54-2
   Cleveland, OH   44135
   216-433-6586

   EMail: mallman@grc.nasa.gov
   http://roland.grc.nasa.gov/˜mallman

   Vern Paxson
   ACIRI / ICSI
   1947 Center Street
   Suite 600
   Berkeley, CA 94704-1198

   Phone: +1 510/642-4274 x302
   EMail: vern@aciri.org

   W. Richard Stevens
   1202 E. Paseo del Zorro
   Tucson, AZ   85718
   520-297-9416

   EMail: rstevens@kohala.com
   http://www.kohala.com/˜rstevens

Full Copyright Statement