TCP Congestion Control
draft-ietf-tcpm-rfc2581bis-06.txt

Status of this Memo

   This Internet-Draft is submitted to IETF in full conformance with
   the provisions of BCP 78 and BCP 79. This document may contain
   material from IETF Documents or IETF Contributions published or made
   publicly available before November 10, 2008. The person(s)
   controlling the copyright in some of this material may not have
   granted the IETF Trust the right to allow modifications of such
   material outside the IETF Standards Process. Without obtaining an
   adequate license from the person(s) controlling the copyright in
   such materials, this document may not be modified outside the IETF
   Standards Process, and derivative works of it may not be created
   outside the IETF Standards Process, except to format it for
   publication as an RFC or to translate it into languages other than
   English.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF), its areas, and its working groups. Note that
   other groups may also distribute working documents as
   Internet-Drafts.

   Internet-Drafts are draft documents valid for a maximum of six
   months and may be updated, replaced, or obsoleted by other documents
   at any time. It is inappropriate to use Internet-Drafts as
   reference material or to cite them other than as "work in progress."

   The list of current Internet-Drafts can be accessed at
   http://www.ietf.org/ietf/1id-abstracts.txt.

   The list of Internet-Draft Shadow Directories can be accessed at
   http://www.ietf.org/shadow.html.

Copyright Statement

rights and restrictions with respect to this document.

This document may contain material from IETF Documents or IETF

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. This document
   obsoletes RFC 2581.

Table Of Contents

1. Introduction

   This document specifies four TCP [RFC793] 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 [RFC1122].
   Additional early work in additive-increase, multiplicative-decrease
   congestion control is given in [CJ89].

   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.

In addition to specifying these 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.

This document obsoletes [RFC2581], which in turn obsoleted
[RFC2001].

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 [RFC2119].

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 [RFC1191,RFC4821] 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 [RFC1122]. 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
          cumulatively acknowledged.

     DUPLICATE ACKNOWLEDGMENT: An acknowledgment is considered a
          "duplicate" in the following algorithms when (a) the receiver of
          the ACK has outstanding data, (b) the incoming acknowledgment
          carries no data, (c) the SYN and FIN bits are both off, (d) the
          acknowledgment number is equal to the greatest acknowledgment
          received on the given connection (TCP.UNA from [RFC793]) and (e)
          the advertised window in the incoming acknowledgment equals the
          advertised window in the last incoming acknowledgment.

          Alternatively, a TCP that utilizes selective acknowledgments
          [RFC2018,RFC2883] can leverage the SACK information to determine
          when an incoming ACK is a "duplicate" (e.g., if the ACK contains
          previously unknown SACK information).

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).

     Also note that the algorithms specified in this document work in
     terms of using loss as the signal of congestion. Explicit
     Congestion Notification (ECN) could also be used as specified in
     [RFC3168].

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. Slow start
additionally serves to start the "ACK clock" used by the TCP sender
to release data into the network in the slow start, congestion
avoidance, and loss recovery algorithms.

IW, the initial value of cwnd, MUST be set using the following
guidelines as an upper bound.

If SMSS > 2190 bytes:
    IW = 2 * SMSS bytes and MUST NOT be more than 2 segments
If (SMSS > 1095 bytes) and (SMSS <= 2190 bytes):
    IW = 3 * SMSS bytes and MUST NOT be more than 3 segments
if SMSS <= 1095 bytes:
    IW = 4 * SMSS bytes and MUST NOT be more than 4 segments

As specified in [RFC3390], the SYN/ACK and the acknowledgment of the
SYN/ACK MUST NOT increase the size of the congestion window.
Further, if the SYN or SYN/ACK is lost, the initial window used by a
sender after a correctly transmitted SYN MUST be one segment
consisting of at most SMSS bytes.

A detailed rationale and discussion of the IW setting is provided in
[RFC3390].

When initial congestion windows of more than one segment are
implemented along with Path MTU Discovery [RFC1191], and the MSS
being used is found to be too large, the congestion window cwnd
SHOULD be reduced to prevent large bursts of smaller segments.
Specifically, cwnd SHOULD be reduced by the ratio of the old segment
size to the new segment size.

The initial value of ssthresh SHOULD be set arbitrarily high (e.g.,
to the size of the largest possible advertised window), but ssthresh
MUST be reduced in response to congestion. Setting ssthresh as high
as possible allows the network conditions, rather than some
arbitrary host limit, to dictate the sending rate. In cases where
the end systems have a solid understanding of the network path, more
carefully setting the initial ssthresh value may have merit (e.g.,
such that the end host does not create congestion along the path).

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 cumulatively acknowledges new data. Slow

start ends when cwnd exceeds ssthresh (or, optionally, when it
reaches it, as noted above) or when congestion is observed. While
traditionally TCP implementations have increased cwnd by precisely
SMSS bytes upon receipt of an ACK covering new data, we RECOMMEND
that TCP implementations increase cwnd, per:

        cwnd += min (N, SMSS)   (2)

where N is the number of previously unacknowledged bytes
acknowledged in the incoming ACK. This adjustment is part of
Appropriate Byte Counting [RFC3465] and provides robustness against
misbehaving receivers which may attempt to induce a sender to
artificially inflate cwnd using a mechanism known as "ACK Division"
[SCWA99]. ACK Division consists of a receiver sending multiple ACKs
for a single TCP data segment, each acknowledging only a portion of
its data. A TCP that increments cwnd by SMSS for each such ACK will
inappropriately inflate the amount of data injected into the
network.

During congestion avoidance, cwnd is incremented by roughly 1
full-sized segment per round-trip time (RTT). Congestion avoidance
continues until congestion is detected. The basic guidelines for
incrementing cwnd during congestion avoidance are:

   * MAY increment cwnd by SMSS bytes

   * SHOULD increment cwnd per equation (2) once per RTT

   * MUST NOT increment cwnd by more than SMSS bytes

We note that [RFC3465] allows for cwnd increases of more than SMSS
bytes for incoming acknowledgments during slow start on an
experimental basis, however such behavior is not allowed as part of
the standard.

The RECOMMENDED 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 SMSS bytes per RTT. This method both allows
TCPs to increase cwnd by one segment per RTT in the face of delayed
ACKs and provides robustness against ACK Division attacks.

Another common formula that a TCP MAY use to update cwnd during
congestion avoidance is given in equation 3:

        cwnd += SMSS*SMSS/cwnd   (3)

This adjustment is executed on every incoming ACK that acknowledges
new data. Equation (3) 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 is

acknowledging every-other packet, (3) is less aggressive than
allowed -- roughly increasing cwnd every second RTT.)

Implementation Note: Since integer arithmetic is usually used in TCP
implementations, the formula given in equation 3 can fail to
increase cwnd when the congestion window is 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 (3). This is
incorrect and can actually lead to diminished performance [RFC2525].

Implementation Note: Some implementations maintain cwnd in units of
bytes, while others in units of full-sized segments. The latter
will find equation (3) 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 and the given segment has not yet been resent by way of the
retransmission timer, the value of ssthresh MUST be set to no more
than the value given in equation 4:

    ssthresh = max (FlightSize / 2, 2*SMSS)   (4)

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

On the other hand, when a TCP sender detects segment loss using the
retransmission timer and the given segment has already been
retransmitted by way of the retransmission timer at least once, the
value of ssthresh is held constant.

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 (as specified in [RFC2988]) 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.

As shown in [FF96,RFC3782], slow start-based loss recovery after a
timeout can cause spurious retransmissions that trigger duplicate
acknowledgments. The reaction to the arrival of these duplicate
ACKs in TCP implementations varies widely. This document does not
specify how to treat such acknowledgments, but does note this as an

area that may benefit from additional attention, experimentation and
specification.

[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 until the loss is
repaired. 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 advanced
loss recovery algorithm, as outlined in section 4.3.

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 (as
defined in section 2, without any intervening ACKs which move
SND.UNA) 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,
since loss is an indication of congestion).

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

1. On the first and second duplicate ACKs received at a sender, a
    TCP SHOULD send a segment of previously unsent data per
    [RFC3042] provided that the receiver's advertised window allows,
    the total FlightSize would remain less than or equal to cwnd

plus 2*SMSS, and that new data is available for transmission.
Further, the TCP sender MUST NOT change cwnd to reflect these
two segments [RFC3042]. Note that a sender using SACK [RFC2018]
MUST NOT send new data unless the incoming duplicate
acknowledgment contains new SACK information.

2. When the third duplicate ACK is received, a TCP MUST set
    ssthresh to no more than the value given in equation 4. When
    [RFC3042] is in use, additional data sent in limited transmit
    MUST NOT be included in this calculation.

3. The lost segment starting at SND.UNA MUST be retransmitted and
    cwnd set 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.

4. For each additional duplicate ACK received (after the third),
    cwnd MUST be incremented by SMSS. This artificially inflates
    the congestion window in order to reflect the additional segment
    that has left the network.

    Note: [SCWA99] discusses a receiver-based attack whereby many
    bogus duplicate ACKs are sent to the data sender in order to
    artificially inflate cwnd and cause a higher than appropriate
    sending rate to be used. A TCP MAY therefore limit the number
    of times cwnd is artificially inflated during loss recovery
    to the number of outstanding segments (or, an approximation
    thereof).

    Note: When an advanced loss recovery mechanism (such as outlined
    in section 4.3) is not in use, this increase in FlightSize can
    cause equation 4 to slightly inflate cwnd and ssthresh, as some
    of the segments between SND.UNA and SND.NXT are assumed to have
    left the network but are still reflected in FlightSize.

5. When previously unsent data is available and the new value of
    cwnd and the receiver's advertised window allow, a TCP SHOULD
    send 1*SMSS bytes of previously unsent data.

6. When the next ACK arrives that acknowledges previously
    unacknowledged data, a TCP MUST set cwnd to ssthresh (the value
    set in step 2). This is termed "deflating" the window.

    This ACK should be the acknowledgment elicited by the
    retransmission from step 3, 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 efficiently
from multiple losses in a single flight of packets [FF96]. Section
4.3 below addresses such cases.

[**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. In addition, changing network conditions may have
rendered TCP's notion of the available end-to-end network capacity
between two endpoints, as estimated by cwnd, inaccurate during the
course of a long 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 = min(IW,cwnd).

Using the last time a segment was received to determine whether or
not to decrease cwnd can fail to deflate cwnd in the common case of
persistent HTTP connections [HTH98]. In this case, a Web server
receives a request before transmitting data to the Web client. 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 [RFC1122] SHOULD be used by a
TCP receiver. When using delayed ACKs, 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 [RFC1122] 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 [RFC2525] 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 [RFC1122], 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, RFC793][RFC813].

4.3 Loss Recovery Mechanisms

A number of loss recovery algorithms that augment fast retransmit
and fast recovery have been suggested by TCP researchers and
specified in the RFC series. While some of these algorithms are
based on the TCP selective acknowledgment (SACK) option [RFC2018],
such as [FF96,MM96a,MM96b,RFC3517], others do not require SACKs
[Hoe96,FF96,RFC3782]. The non-SACK algorithms use "partial
acknowledgments" (ACKs which cover previously unacknowledged 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.

That is, when the first loss in a window of data is detected,

ssthresh MUST be set to no more than the value given by equation
(4). 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.

We RECOMMEND that TCP implementers employ some form of advanced loss
recovery that can cope with multiple losses in a window of data.
The algorithms detailed in [RFC3782] and [RFC3517] conform to the
general principles outlined above. We note that while these are not
the only two algorithms that conform to the above general principles
these two algorithms have been vetted by the community and are
currently on the standards track.

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.

In response to the ACK division attack outlined in [SCWA99] this
document RECOMMENDS increasing the congestion window based on the
number of bytes newly acknowledged in each arriving ACK rather than
by a particular constant on each arriving ACK (as outlined in
section 3.1).

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 Between RFC 2001 and RFC 2581

[RFC2001] was extensively rewritten editorially and it is not
feasible to itemize the list of changes between [RFC2001] and
[RFC2581]. The intention of [RFC2581] was to not change any of the
recommendations given in [RFC2001], but to further clarify cases
that were not discussed in detail in [RFC2001]. Specifically,
[RFC2581] suggested what TCP connections should do after a
relatively long idle period, as well as specified and clarified
some of the issues pertaining to TCP ACK generation. Finally, the
allowable upper bound for the initial congestion window was raised

from one to two segments.

7. Changes Relative to RFC 2581

A specific definition for "duplicate acknowledgment" has been

added, based on the definition used by BSD TCP.

The document now notes that what to do with duplicate ACKs after the retransmission timer has fired is future work and explicitly unspecified in this document.

The initial window requirements were changed to allow Larger Initial Windows as standardized in [RFC3390]. Additionally, the steps to take when an initial window is discovered to be too large due to Path MTU Discovery [RFC1191] are detailed.

The recommended initial value for ssthresh has been changed to say that it SHOULD be arbitrarily high, where it was previously MAY. This is to provide additional guidance to implementors on the matter.

During slow start, the usage of Appropriate Byte Counting [RFC3465] with L=1*SMSS is explicitly recommended. The method of increasing cwnd given in [RFC2581] is still explicitly allowed. Byte counting during congestion avoidance is also recommended, while the method from [RFC2581] and other safe methods are still allowed.

The treatment of ssthresh on retransmission timeout was clarified. In particular, ssthresh must be set to half the FlightSize on the first retransmission of a given segment and then is held constant on subsequent retransmissions of the same segment.

The description of fast retransmit and fast recovery has been clarified, and the use of Limited Transmit [RFC3042] is now recommended.

TCPs now MAY limit the number of duplicate ACKs that artificially inflate cwnd during loss recovery to the number of segments outstanding to avoid the duplicate ACK spoofing attack described in [SCWA99].

The restart window has been changed to min(IW,cwnd) from IW. This behavior was described as "experimental" in [RFC2581].

It is now recommended that TCP implementors implement an advanced loss recovery algorithm conforming to the principles outlined in this document.

The security considerations have been updated to discuss ACK division and recommend byte counting as a counter to this attack.

8. IANA Considerations

This document contains no IANA considerations, but apparently an Internet *Draft* can no longer be published without this section.

Acknowledgments

   The core algorithms we describe were developed by Van Jacobson

[Jac88, Jac90]. In addition, Limited Transmit [RFC3042] was
developed in conjunction with Hari Balakrishnan and Sally Floyd.
The initial congestion window size specified in this document is a
result of work with Sally Floyd and Craig Partridge
[RFC2414,RFC3390].

W. Richard ("Rich") Stevens wrote the first version of this document
[RFC2001] and co-authored the second version [RFC2581]. This
present version much benefits from his clarity and thoughtfulness of
description, and we are grateful for Rich's contributions in
elucidating TCP congestion control, as well as in more broadly
helping us understand numerous issues relating to networking.

We wish to emphasize that the shortcomings and mistakes of this
document are solely the responsibility of the current authors.

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.

Anil Agarwal, Steve Arden, Neal Cardwell, Noritoshi Demizu, Gorry
Fairhurst, Kevin Fall, John Heffner, Alfred Hoenes, Sally Floyd,
Reiner Ludwig, Matt Mathis, Craig Partridge and Joe Touch
contributed a number of helpful suggestions.

Normative References

   [RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC
      793, September 1981.

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

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

Informative References

   [CJ89] Chiu, D. and R. Jain, "Analysis of the Increase/Decrease
      Algorithms for Congestion Avoidance in Computer Networks",
      Journal of Computer Networks and ISDN Systems, vol. 17, no. 1,
      pp. 1-14, June 1989.

   [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.

[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.

   [MM96a] Mathis, M. and J. Mahdavi, "Forward Acknowledgment: Refining
        TCP Congestion Control", Proceedings of SIGCOMM'96, August,
        1996, Stanford, CA. Available
        from http://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.

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

   [RFC813] Clark, D., "Window and Acknowledgment Strategy in TCP", RFC
        813, July 1982.

   [RFC2001] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast
        Retransmit, and Fast Recovery Algorithms", RFC 2001, January
        1997.

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

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

   [RFC2414] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's
        Initial Window Size", RFC 2414, September 1998.

   [RFC2525] 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.

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

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

   [RFC2988] V. Paxson and M. Allman, "Computing TCP's Retransmission

          Timer", RFC 2988, November 2000.

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

2001.

[RFC3168] K. Ramakrishnan, S. Floyd, D. Black, "The Addition of
          Explicit Congestion Notification (ECN) to IP", RFC 3168,
          September 2001.

[RFC3390] Allman, M., Floyd, S., C. Partridge, "Increasing TCP's
          Initial Window", RFC 3390, October 2002.

[RFC3465] Mark Allman, TCP Congestion Control with Appropriate Byte
          Counting (ABC), RFC 3465, February 2003.

[RFC3517] Ethan Blanton, Mark Allman, Kevin Fall, Lili Wang, A
          Conservative Selective Acknowledgment (SACK)-based Loss Recovery
          Algorithm for TCP, RFC 3517, April 2003.

[RFC3782] Sally Floyd, Tom Henderson, Andrei Gurtov, The NewReno
          Modification to TCP's Fast Recovery Algorithm, RFC 3782, April
          2004.

[RFC4821] Matt Mathis, John Heffner, Packetization Layer Path MTU
          Discovery, RFC 4821, March 2007.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson,
          "TCP Congestion Control With a Misbehaving Receiver", ACM
          Computer Communication Review, 29(5), October 1999.

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

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

Authors' Addresses

    Mark Allman
    International Computer Science Institute (ICSI)
    1947 Center Street
    Suite 600
    Berkeley, CA 94704-1198
    Phone: +1 440 235 1792
    EMail: mallman@icir.org
    http://www.icir.org/mallman/

    Vern Paxson
    International Computer Science Institute (ICSI)
    1947 Center Street
    Suite 600
    Berkeley, CA 94704-1198

       Phone: +1 510/642-4274 x302
       EMail: vern@icir.org
       http://www.icir.org/vern/

Ethan Blanton
Purdue University Computer Sciences
305 North University Street
West Lafayette, IN 47907
EMail: eblanton@cs.purdue.edu
http://www.cs.purdue.edu/homes/eblanton/

Acknowledgment