Detection and Quantification of Packet Reordering with TCP
draft-zimmermann-tcpm-reordering-detection-01

Abstract

   This document specifies an algorithm for the detection and
   quantification of packet reordering for TCP. In the absence of
   explicit congestion notification from the network, TCP uses only
   packet loss as an indication of congestion. One of the signals TCP
   uses to determine loss is the arrival of three duplicate
   acknowledgments. However, this heuristic is not always correct,
   notably in the case when paths reorder packets. This results in
   degraded performance.

   The algorithm for the detection and quantification of reordering in
   this document uses information gathered from the TCP Timestamps
   Option, the TCP SACK Option and its DSACK extension. When a
   reordering event is detected, the algorithm calculates a reordering
   extent by determining the number of segments the reordered segment
   was late with respect to its position in the sequence number space.
   Additionally, the algorithm computes a second reordering extent that
   is relative to the amount of outstanding data and thus provides a
   better estimation of the reordering delay when other sender state
   changes.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   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."

This Internet-Draft will expire on November 21, 2014.

Copyright Notice

Table of Contents

1. Introduction

   When the Transmission Control Protocol (TCP) [RFC0793] decides that
   the oldest outstanding segment is lost, it performs a retransmission
   and changes the sending rate [RFC5681]. This occurs either when the
   Retransmission Timeout (RTO) timer expires for a segment [RFC6298],
   or when three duplicate acknowledgments (ACKs) for a segment have
   been received (Fast Retransmit/Fast Recovery) [RFC5681]. The
   assumption behind Fast Retransmit is that non-congestion events that
   can cause duplicate ACKs to be generated (packet duplication, packet
   reordering and packet corruption) are infrequent. However, a number
   of Internet measurement studies have shown that packet reordering is
   not a rare phenomenon [Pax97], [BPS99], [BS02], [ZM04], [GPL04],
   [Jai+07] and has negative performance implications on TCP [BA02],
   [Zha+03].

   The impact that packet reordering has on TCP can be classified by the
   type of reordering: forward-path versus reverse-path reordering.

   From TCP's perspective, the result of packet reordering on the
   forward-path is the reception of out-of-order segments by the TCP
   receiver. In response to every received out-of-order segment, the
   TCP receiver immediately sends a duplicate ACK. (Note: [RFC5681]
   recommends that delayed ACKs not be used when the ACK is triggered by
   an out-of-order segment.) The sender side, if the number of
   consecutively received duplicate ACKs exceeds the duplicate
   acknowledgment threshold (DupThresh), retransmits the first
   unacknowledged segment [RFC5681] and continues with a loss recovery
   algorithm such as NewReno [RFC6582] or the Selective Acknowledgment
   (SACK) based loss recovery [RFC6675]. If a segment arrives due to
   reordering more than three segments (the default value of DupThresh
   [RFC5681]) too late at the TCP receiver, the sender is not able to
   distinguish this reordering event from a segment loss, resulting in
   an unnecessary retransmission and rate reduction.

   Packet reordering may not only cause data segments to arrive out-of-
   order at the receiver but also ACKs ot the sender. This reordering
   on the reverse path also has a negative an impact on TCP performance,
   by causing a degradation of TCP's self-clocking property. In steady
   state, depending on whether the TCP receiver delays an ACK or not
   [RFC1122], one or two segments are acknowledged per ACK. If, due to
   reordering on the reverse path, ACKs arrive at the TCP sender in a
   different order than they were sent in by the TCP receiver, in-order
   ACKs acknowledge several segments together rather than only one or
   two, while disordered ACKs arrive either out-of-order or out-of-
   window and are ignored. (Note: according to [RFC6675], an ACK only
   counts as a duplicate if it carries a SACK block that identifies
   previously unacknowledged and un-SACKed data.) Overall, this leads

to a bursty transmission pattern as well as outdated SACK and DSACK information.

Since DupThresh is defined in segments rather than bytes [RFC5681], TCP usually quantifies packet reordering in terms of segments. Informally, the reordering extent [RFC4737] is defined as the maximum distance in segments between the reception of a reordered segment and the earliest segment received with a larger sequence number. If a segment is received in-order, its reordering extent is undefined [RFC4737]. On the basis of the reordering extent, a simple robustness of TCP to packet reordering can be achieved by directly applying the reordering extent as DupThresh. A problem that arises with this way of quantifying reordering is that even in the presence of constant reordering, reordering extents may vary if the transmission rate of the TCP sender changes. Therefore, by using a DupThresh that directly reflects the measured reordering extent, spurious retransmissions cannot be fully avoided.

The following example illustrates this issue. Assume a path with a reordering probability of 1%, a reordering delay of 20 ms, and a bottleneck bandwidth of 3 Mb/s. Because segments that are delayed by reordering arrive 20 ms too late, the TCP receiver can receive a maximum of $((20 * 3 * 10^3) / 8) = 7500$ bytes out-of-order before the reordered segment arrives. Hence, with a Sender Maximum Segment Size (SMSS) of 1460 bytes, the largest possible reordering extent is close to 5 segments. If the bottleneck bandwidth changes from 3 Mb/s to 4 Mb/s, the maximum reordering extent will increase to 7 segments, although the reordering delay remains constant.

This simple example shows that even with constant reordering, spurious retransmissions cannot be completely avoided if the DupThresh directly reflects the reordering extent. On the other hand, the reordering extent and the resulting DupThresh can sometimes also be much too high and do not correspond to the actual packet reordering present on the path. For example, a slow start overshoot [Hoe96], [MM96], [Mat+97] at the end of slow start might induce such a problem.

An obvious solution to the problem would be to quantify packet reordering not by calculating a reordering extent, but by using the reordering late time offset [RFC4737]. Since the reordering late time offset is not specified in segments but captures the difference between the expected and actual reception time of a reordered segment, this way of quantifying reordering is independent of the current transmission rate. Disadvantages of this approach are however a higher complexity and a worse integration into the TCP specification, since an implementation would require additional timers, whereas TCP itself is self-clocked.

The approach taken by this specification quantifies the reordering
extend for a packet not only through an absolute value, but also
through a measure that is relative to the amount of outstanding data,
in an attempt to approximate a time-based measure. The presented
scheme can thereby easily be adapted to the Stream Control
Transmission Protocol (SCTP) [RFC2960], since SCTP uses congestion
control algorithms similar to TCP.

The remainder of this document is organized as follows. Section 3
provides a high-level description of the packet reordering detection
mechanisms. In Section 4, the algorithm is specified. In Section 5,
each step of the algorithm is discussed in detail. Section 6
provides a discussion of several design decisions of the algorithm.
Section 7 discusses related work. Section 9 discusses security
concerns.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described [RFC2119].

The reader is expected to be familiar with the TCP state variables
given in [RFC0793] (SEG.SEQ, SND.UNA), [RFC5681] (FlightSize), and
[RFC6675] (DupThresh, SACK scoreboard). SND.FACK (forward
acknowledgment) is used to describe the highest sequence number -
plus one - that has been either cumulatively or selectively
acknowledged by the receiver and subsequently seen by the sender
[MM96]. Further, the term 'acceptable acknowledgment' is used as
defined in [RFC0793]. That is, an ACK that increases the
connection's cumulative ACK point by acknowledging previously
unacknowledged data. The term 'duplicate acknowledgment' is used as
defined in [RFC6675], which is different from the definition of
duplicate acknowledgment in [RFC5681].

This specification defines the four TCP sender states 'open',
'disorder', 'recovery', and 'loss' as follows. As long as no
duplicate ACK is received and no segment is considered lost, the TCP
sender is in the 'open' state. Upon the reception of the first
consecutive duplicate ACK, TCP will enter the 'disorder' state.
After receiving DupThresh duplicate ACKs, the TCP sender switches to
the 'recovery' state and executes standard loss recovery procedures
like Fast Retransmit and Fast Recovery [RFC5681]. Upon a
retransmission timeout, the TCP sender enters the 'loss' state. The
'recovery' state can only be reached by a transition from the
'disorder' state, the 'loss' state can be reached from any other
state.

3. Basic Concepts

   The following specification depends on the TCP Timestamps option
   [RFC1323], the TCP Selective Acknowledgment (SACK) [RFC2018] option
   and the latter's Duplicate Selective Acknowledgment (DSACK) extension
   [RFC2883]. The reader is assumed to be familiar with the algorithms
   specified in these documents.

   Reordering is quantified by an absolute and a relative reordering
   extent. If a hole in the SACK scoreboard of the TCP sender is closed
   either cumulatively by an acceptable ACK or selectively by a new
   SACK, then the absolute reordering extent is computed as the number
   of segments in the SACK scoreboard between the sequence number of the
   reordered segment and the highest selectively or cumulatively
   acknowledged sequence number. The relative reordering extent is then
   computed as the ratio between the absolute reordering extent and the
   FlightSize stored when entering the 'disorder' state.

   If the hole that was closed in the SACK scoreboard corresponds to a
   segment that was not retransmitted, or if the retransmission of such
   a segment can be determined as a spurious retransmission by means of
   the Eifel detection algorithm [RFC3522], then the calculated
   reordering extent is immediately valid. Otherwise, if the
   verification of the Eifel detection algorithm has not been possible,
   the reordering extent is stored for a possibly subsequent DSACK. If
   no such DSACK is received in the next two round-trip times (RTTs),
   the reordering extents are discarded.

4. The Algorithm

   Given that usually both the Nagle algorithm [RFC0896] [RFC1122] and
   the TCP Selective Acknowledgment Option [RFC2018] are enabled, a TCP
   sender MAY employ the following algorithm to detect and quantify the
   current perceived packet reordering in the network.

   Without the Nagle algorithm, there is no straight forward way to
   accurately calculate the number of outstanding segments in the
   network (and, therefore, no good way to derive an appropriate
   reordering extent) without adding state to the TCP sender. A TCP
   connection that does not employ the Nagle algorithm SHOULD NOT use
   this methodology.

   If a TCP sender implements the following algorithm, the
   implementation MUST follow the various specifications provided in
   Sections 4.1 to 4.8. The algorithm MUST be executed *before* the
   Transmission Control Block or the SACK scoreboard have been updated
   by another loss recovery algorithm.

4.1. Initialization During Connection Establishment

   After the completion of the TCP connection establishment, the
   following state variables MUST be initialized in the TCP transmission
   control block:

   (C.1) The variable Dsack, which indicates whether a DSACK has been
          received so far, and the data structure Samples, which stores
          the computed reordering extents, MUST be initialized as:

               Dsack = false
               Samples = []

   (C.2) If the TCP Timestamps option [RFC1122] has been negotiated,
          then the variable Timestamps MUST be activated and the data
          structure Retrans_TS, which stores the value of the TSval
          field of the retransmissions sent during Fast Recovery, MUST
          be initialized as:

               Timestamps = true
               Retrans_TS = []

          Otherwise, the Timestamps-based detection SHOULD be
          deactivated:

               Timestamps = false

4.2. Receiving Acknowledgments

   For each received ACK that either a) carries SACK information, *or*
   b) is a full ACK that terminates the current Fast Recovery procedure,
   or c) is an acceptable ACK that is received immediately after a
   duplicate ACK, execute steps (A.1) to (A.4), otherwise skip to step
   (A.4).

   (A.1) If a) the ACK carries new SACK information, *and* b) the SACK
          scoreboard is empty (i.e., the TCP sender has received no SACK
          information from the receiver), then the TCP sender MUST save
          the amount of current outstanding data:

               FlightSizePrev = FlightSize

   (A.2) If the received ACK either a) cumulatively acknowledges at
          most SMSS bytes, *or* b) selectively acknowledges at most SMSS
          bytes in the sequence number space in the SACK scoreboard,
          then:

The TCP sender MUST execute steps (S.1) to (S.4)

(A.3) If a) Timestamps == false *and* b) the received ACK carries a
       DSACK option [RFC2883] and the segment identified by the DSACK
       option can be marked according to step (A.1) to (A.4) of
       [RFC3708] as a valid duplicate, then:

           The TCP sender MUST execute steps (D.1) to (D.3)

(A.4) The TCP sender MUST terminate the processing of the ACK by
       this algorithm and MUST continue with the default processing
       of the ACK.

4.3. Receiving Acknowledgment Closing Hole

(S.1) If (a) the newly cumulatively or selectively acknowledged
       segment SEG is a retransmission *and* b) both equations Dsack
       == false and Timestamps == false hold, then the TCP sender
       MUST skip to step (A.4).

(S.2) Compute the relative and absolute reordering extent ReorExtR,
       ReorExtA:

           The TCP sender MUST execute steps (E.1) to (E.4)

(S.3) If a) the newly acknowledged segment SEG was not retransmitted
       before *or* b) both equations Timestamps == true and
       Retrans_TS[SEG.SEQ] > ACK.TSecr hold, i.e., the ACK
       acknowledges the original transmission and not a
       retransmission, then hand over the reordering extents to an
       additional reaction algorithm:

           The TCP sender MUST execute step (P)

(S.4) If a) the previous step (S.3) was not executed *and* b) both
       equations Dsack == true and Timestamps == false hold, save the
       reordering extents for the newly acknowledged segment SEG for
       at least two RTTs:

           Samples[SEG.SEQ].ReorExtR = ReorExtR
           Samples[SEG.SEQ].ReorExtA = ReorExtA

4.4. Receiving Duplicate Selective Acknowledgment

(D.1) If no DSACK has been received so far, the sender MUST set:

           Dsack = true

(D.2) If a) the previous step (D.1) was not executed *and* a
       reordering extent was calculated for the segment SEG
       identified by the DSACK option, then the TCP sender MUST
       restore the values of the variables ReorExtR and ReorExtA and
       delete the corresponding entries in the data structure:

           ReorExtR = Samples[SEG.SEQ].ReorExtR
           ReorExtA = SAMPLES[SEG.SEQ].ReorExtA

(D.3) Hand the newly restored reordering extents over to an
       additional reaction algorithm:

           The TCP sender SHOULD execute step (P)

4.5. Reordering Extent Computation

(E.1) SEG.SEQ is the sequence number of the newly cumulatively or
       selectively acknowledged segment SEG.

(E.2) SND.FACK is the highest either cumulatively or selectively
       acknowledged sequence number so far plus one.

(E.3) The TCP sender MUST compute the absolute reordering extent
       ReorExtA as

           $ReorExtA = (SND.FACK - SEG.SEQ) / SMSS$

(E.4) The TCP sender MUST compute the relative reordering extent
       ReorExtR as

           $ReorExtR = ReorExtA * (SMSS / FlightSizePrev)$

4.6. Retransmitting Segment

   If the TCP Timestamps option [RFC1323] is used to detect packet
   reordering, the TCP sender must save the TCP Timestamps option of all
   retransmitted segments during Fast Recovery.

(RET) If a) a segment SEG is retransmitted during Fast Recovery,
       and b) the equation Timestamps = true holds, the TCP sender
       MUST save the value of the TSval field of the retransmitted
       segment:

              Retrans_TS[SEG.SEQ] = SEG.TSval

4.7. Placeholder for Response Algorithm

   (P)   This is a placeholder for an additional reaction algorithm
          that takes further action using the results of this algorithm,
          for example, the adjustment of the DupThresh based on relative
          and absolute reordering extent ReorExtR and ReorExtA.

4.8. Retransmission Timeout

   The expiration of the retransmission timer should be interpreted as
   an indication of a change in path characteristics, and the TCP sender
   should consider all saved reordering extents as outdated and delete
   them.

   (RTO) If an retransmission timeout (RTO) occurs, a TCP sender SHOULD
          reset the following variables:

                  Samples = []
                  Retrans_TS = []

5. Protocol Steps in Detail

   The reception of an ACK represents the starting point for the
   detection scheme above. For each received SACK, DSACK or acceptable
   ACK that prompts the TCP sender to enter the 'disorder' state, to
   remain in the 'disorder' state or to leave either the 'disorder' or
   'recovery' states towards the 'open' state, steps (A.1) to (A.4) are
   performed. All other received ACKs are not relevant for the
   detection of packet reordering and can be ignored. If the TCP sender
   changes from the 'open' to the 'disorder' state due to the reception
   of a duplicate ACK (i.e., the SACK scoreboard is empty and an ACK
   arrives carrying new SACK information), the current amount of
   outstanding data, FlightSize, is stored for the subsequent
   calculation of the relative reordering extent (step (A.1)).

   Whenever a received acceptable ACK or SACK closes a hole in the
   sequence number space of the SACK scoreboard either partially or
   completely, this is an indication of packet reordering in the network
   (step (A.2)). The prerequisite for an accurate quantification of the
   reordering is that only one segment is newly acknowledged (maximum
   SMSS bytes of data). If more than one segment per ACK is
   acknowledged, either by reordering on the reverse path or the loss of
   ACKs, the order in which the segments have been received by the TCP
   receiver is no longer accurately determinable so that in this case a
   reordering extent is not calculated. Finally, if the received ACK

carries a DSACK option that identifies a segment that was
retransmitted only once, then this is sufficient to conclude
reordering (step (A.3)), so that a previously calculated reordering
extent can be passed to another algorithm (steps (D.3) and (P)).

With just the information provided by the ACK field or SACK
information above SND.UNA, the TCP sender is unable to distinguish
whether the ACK that finally acknowledges retransmitted data (either
cumulatively or selectively) was sent in response to the original
segment or a retransmission of the segment. This is described as the
retransmission ambiguity problem in [KP87]. Therefore, the detection
and quantification of reordering depends on other means to
distinguish between acknowledgments for transmission and
retransmission to detect if a retransmission was spurious. If
neither a DSACK has been received (Dsack == false) so far nor the TCP
Timestamps option has been enabled on connection establishment
(Timestamps == false) then there is no possibility for the TCP sender
to identify spurious retransmissions. Hence, the processing of the
received ACK by the detection algorithm must be terminated for
retransmitted segments (step (S.1)). Otherwise, if the segment that
corresponds to the closed hole in the sequence number space of the
SACK scoreboard has not been retransmitted or the retransmission can
be identified by the Eifel detection algorithm [RFC3522] as a
spurious retransmission, the previously calculated reordering extent
is valid (step (S.2)) and an additional reaction algorithm can be
executed (steps (S.3) and (P)).

For the use of the Eifel detection it is necessary to store the TCP
Timestamps option of all retransmissions sent during Fast Recovery
(step (Ret)). However, if the use of the Eifel detection algorithm
is not possible (Timestamps == false), the extent of a possible
reordering is stored for the possibility of a subsequent arrival of a
DSACK (step (P.4)). If no such DSACK is received in the next two
round-trip times, the reordering extent is discarded. Since the
DSACK extension is not negotiated during connection establishment
[RFC2883], the reordering extent is only stored if a DSACK was
previously received for the TCP connection (DSACK == true, step
(D.1)).

Regardless of whether packet reordering is detected by using the
SACK-based methodology, the DSACK-based methodology, or the TCP
Timestamps option, quantification of the reordering will always be
done when closing a hole in the sequence number space of the SACK
scoreboard (step (A.2), step (P.2)). The only difference is the time
of detection, which is in the case of DSACK-based methodology at
least one RTT after the time of the quantification. The absolute
reordering extent ReorExtA results from the number of segments in the
SACK scoreboard between the sequence number of the newly acknowledged

segment and the highest either cumulatively or selectively
acknowledged sequence number so far plus one (SND.FACK) (step (E.3)).

It is worth noting that the absolute reordering extent includes all
segments (bytes) between the closed hole and the highest acknowledged
sequence number so far, i.e., it also includes segments (bytes) that
are not selectively acknowledged. The reason is that if packet
reordering is considered from a temporal perspective, it is
irrelevant whether there are lost segments or not. The important
fact is that the lost segments have been sent after the delayed
segment and before the highest acknowledged segment, which is
expressed by the metric. In step (E.4), the relative reordering
extent ReorExtR is then calculated by the ratio between the absolute
reordering extent ReorExtA and the amount of outstanding data stored
by step (A.1).

6. Discussion of the Algorithm

The focus of the following discussion is on the quantification of
reordering by the relative reordering extent and to elaborate on
possible sources of error, which may lead to an inaccurate detection
and quantification of reordering in the network.

6.1. Calculation of the Relative Reordering Extent

Generally, the characteristics of a relative reordering extent should
be that if packet reordering on a path is constant in terms of rate
and delay, the relative reordering extent should also be constant,
regardless of the current transmission rate of the TCP sender. The
scheme proposed in this document is to calculate the relative
reordering by getting the ratio between absolute reordering (the
amount of data the reordered segment was received too late) and the
amount of outstanding data stored when TCP sender was entering the
'disorder' state (the maximum amount of data a reordered segment can
be received too late). (Note: A segment can theoretically be delayed
for an arbitrarily long period during reordering. However, the
scheme proposed in this document does not capture such kind of
reordering. See Section 6.2.) Therefore, the relative reordering
extent expresses the portion of currently outstanding data that is
selectively acknowledged before the reordered segment is cumulatively
acknowledged. If the transmission rate changes, the absolute
reordering extent changes as well, but together with the amount of
outstanding data, and hence the relative reordering extent stays
constant.

A characteristic of the calculation of the relative reordering extent
on the basis of currently outstanding amount of data is that the

FlightSize reflects the bandwidth-delay-product and not the
transmission rate. As a consequence, the relative reordering extent
is not independent of the RTT. If the RTT of the communication path
changes, the amount of outstanding data changes as well, but the
absolute reordering extent remains constant. Hence, the relative
reordering extent adapts. In principle it is possible to design an
algorithm to compute the relative reordering extent independently of
the RTT and to reflect only the characteristics of packet reordering
of the path. But since the calculation would be far from trivial and
introducing more complexity, this is considered to be future
research.

6.2. Reordering Delay Longer than RTT

   The quantification of packet reordering always takes place at the
   time when a hole in sequence number space in the SACK scoreboard is
   closed. In consequence, if a reordered packet is delayed by more
   than the measured RTT, the amount of reordering on the path is
   underestimated, since the ACK that closes the hole in the sequence
   number space was not sent in response to the original transmission,
   but to the retransmission. Hence, in order to detect and quantify
   these kinds of events, the reordering extent would have to be
   calculated when the ACK for the transmission is received and not at
   the time the hole in the sequence number space is closed. Although
   it would be possible to take these events into account by evaluating
   DSACKs and the TCP Timestamps option simultaneously, the scheme
   proposed in this document refrains from qualifying a reordering delay
   longer then RTT. A reordering delay of this magnitude is very
   unlikely, and would lead to a significant overhead in memory usage
   and complexity. Additionally, taking it into account would result in
   other problems, especially a potential expiration of the RTO
   [I-D.zimmermann-tcpm-reordering-reaction].

6.3. Persistent Reception of Selective Acknowledgments

   Especially on paths with a high bandwidth-delay-product, it is
   possible that even with a minor packet reordering, several segments
   in a single window of data are delayed. If, in addition, the
   sequence numbers of those segments are widely spaced in the sequence
   number space and the delay caused by packet reordering is
   sufficiently high, this might lead to a constant reception of out-of-
   order data. Hence, for each received segment, regardless of whether
   a hole in the sequence number space of the receive window is closed
   or not, an ACK is sent that carries SACK information. From TCP
   sender's perspective, this persistent receiving of new SACK
   information leads to the situation that the TCP sender enters the
   'disorder' state when receiving the first SACK and never leaves it
   again during the connection lifetime if no segment is lost in

between.

In case of the above reordering detection and quantification scheme, the persistent reception of SACK blocks causes the amount of outstanding data, which is stored when the TCP sender enters the 'disorder' state, to never be updated, since FlightSize is only saved in step (A.1) when the SACK scoreboard is empty. If the transmission rate of the TCP sender, and therefore also the maximum amount of data a reordered segment can be received too late, changes significantly during its stay in the 'disorder' state, the actual amount of reordering is not accurately determined by the relative reordering extent. A decrease of the transmission rate would result in an overestimation of the reordering extent and vice versa.

A simple solution to the problem would be to store the maximum offset in terms of sequence number space by which a reordered segment can be received too late only when entering the 'disorder' state, but individually for every potentially reordered segment, that is, for every hole in the sequence number space of the SACK scoreboard. (Note: The maximum offset in terms of sequence number space by which a reordered segment can be received too late is strictly speaking the amount of data that have been transmitted later than the reordered segment. This amount of data can only be expressed by FlightSize within the 'open' state and not within the 'disorder' state, since the cumulative ACK point may not advance).

The problem with this simple idea is that for a new hole in the SACK scoreboard, it is not possible to determine whether it is a result of packet reordering or loss, and therefore it results in increased memory usage (to store the amount of data for each hole). Additionally, the packet reordering would be inaccurately quantified if the transmission rate changes significantly for a short amount of time. For example, if the amount of outstanding data is low when entering the 'disorder' state is entered, the execution of Careful Extended Limited Transmit (as described in [I-D.zimmermann-tcpm-reordering-reaction] [RFC4653]) leads to a significant short-term change of the transmission rate. When the amount of data by which the reordering segment can be delayed is determined individually for every new hole, it leads to an overestimation of the relative reordering extent, since the maximum amount of data possible is 'artificially' reduced by Careful Extended Limited Transmit.

A solution to this problem is to store the maximum offset in terms of sequence number space by which a reordered segment can be received too late not for every segment individually (which does not guarantee an accurate calculation of the relative reordering extent) but only sufficiently often, e.g., once per RTT. The identification of what

frequency would be adequate, though, is neither trivial nor
universally applicable, since a concrete solution depends on the
transmission behavior of the used TCP in the 'disorder' state and
whether it is more beneficial for an additional reordering response
algorithm to over- or underestimate the packet reordering on the
path. If, for example, TCP-aNCR
[I-D.zimmermann-tcpm-reordering-reaction] is used as additional
reordering response algorithm, the maximum offset in terms of
sequence number space by which a reordered segment can be received
too late is not only stored when entering the 'disorder' state but
also updated every RTT (every cwnd worth of data transmitted without
a loss) while the TCP sender stays in the 'disorder' state.

6.4. Packet Duplication

   Although the problem of packet duplication in today's Internet
   [Jai+07], [Mel+08] is negligible, it may happen in rare cases that
   segments on the path to the TCP receiver are duplicated. If a
   segment is duplicated on the path, the first incoming segment causes
   the receiver to send either an acceptable ACK or a SACK, depending on
   whether the segment is the next expected one or not. Each subsequent
   identical segment then causes either a duplicate ACK or a DSACK,
   respectively, depending on whether the DSACK extension [RFC3708] is
   implemented or not.

   If by a combination of packet loss and packet duplication the case
   occurs that a Fast Retransmit for a lost segment is duplicated on the
   path, the TCP sender is not able to distinguish this from packet
   reordering. The first received ACK closes a hole in the sequence
   number space of the SACK scoreboard, while the second received ACK is
   a valid DSACK. Although both cases are indistinguishable from a
   theoretical point of view, the TCP sender can take measures to ensure
   as far as possible that the DSACK received was not the result of
   packet duplication.

   For this purpose, step (A.3) of the above detection method checks via
   the steps (A.1) to (A.4) of [RFC3708] whether the segment identified
   by the DSACK option is marked as a valid duplicate. Unfortunately,
   the steps of [RFC3708] do not check that more DSACKs have been
   received than retransmissions have been sent, which is a
   characteristic of suffering both packet reordering and packet
   duplication at the same time. By simply counting the received
   DSACKs, for example, as additional step (A.5) in [RFC3708], this
   corner case can be covered as well.

7. Related Work

   Because of retransmission ambiguity problem [KP87], which describes
   TCP sender's inability to to distinguish whether the first acceptable
   ACK that arrives after a retransmit was sent in response to the
   original transmit or the retransmit, two different approaches can
   generally be taken to detect and quantify packet reordering. First,
   for transmissions (non-retransmitted segments), the detection is
   usually conducted by detecting a closed hole in sequence number space
   of the SACK scoreboard. Second, for retransmissions, the detection
   of packet reordering is accompanied by the detection of the spurious
   Fast Retransmits.

   Within the IETF, several proposals have been published in the RFC
   series to detect and quantify packet reordering. With [RFC4737] the
   IPPM Working Group [IPPM] defines several metrics to evaluate whether
   a network path has maintained packet order on a packet-by-packet
   basis. [RFC4737] gives concrete, well-defined metrics, along with a
   methodology for applying the metric to a generic packet stream. The
   metric discussed in this document has a different purpose from the
   IPPM metrics; this document discusses a TCP specific reordering
   metric calculated on the TCP sender's SACK scoreboard.

   Besides the IPPM work, several other proposals have been developed to
   detect spurious retransmissions with TCP. The Eifel detection
   algorithm [RFC3522] uses the TCP Timestamps option to determine
   whether the ACK for a given retransmit is for the original
   transmission or a retransmission. The F-RTO scheme [RFC5682]
   slightly alters TCP's sending pattern immediately following a
   retransmission timeout to indicate whether the retransmitted segment
   was needed. Finally, the DSACK-based algorithm [RFC3708] uses the
   TCP SACK option [RFC2018] with the DSACK extension [RFC2883] to
   identify unnecessary retransmissions. The mechanism for detecting
   packet reordering outlined in this document rely on the detection
   schemes of those documents (except F-RTO that only works for spurious
   retransmits triggered by TCP's retransmission timer), although they
   do not provide metrics for the reordering extent whereas the
   algorithm described in this document does.

   RR-TCP [Zha+03] describes a reordering detection and quantification
   scheme that is also based on holes in the sequence number space of
   the SACK scoreboard and the reception of DSACKs. For every hole in
   the SACK scoreboard, RR-TCP calculates a reordering extent. If the
   segment was retransmitted before an ACK was received, it waits for a
   DSACK that proves that the segment was spuriously retransmitted. The
   reordering sample in such a case is the mean between the sample
   calculated due to the hole in the sequence number space and the
   sample calculated in responding to the received DSACK. In contrast,

the methodology in this document is always to quantify the reordering at the time of a closed hole and to not take packet reordering with a delay larger than one RTT into account (see Section 6.2.

The Linux kernel [Linux] implements a reordering detection based on SACK, DSACK and TCP Timestamps option as well. The detection and quantification of non-retransmitted segments with SACK or for retransmitted segments with TCP Timestamps option operates much like the scheme described in this document, with the exception of the DSACK detection. First, Linux does not store any information (e.g., reordering extent) below the cumulative ACK point, so that DSACKs below the cumulative ACK point are ignored (for the purpose for reordering quantification). Second, Linux also does not store any information about a possible reordering event when a hole in the sequence number space of the SACK scoreboard is closed. Therefore, for a DSACK reporting a duplicate above the cumulative ACK, Linux needs to approximate the reordering on arrival of a DSACK by the distance between the DSACK and the highest selectively acknowledged segment.

8. IANA Considerations

   This memo includes no request to IANA.

9. Security Considerations

   The described algorithm neither improves nor degrades the current security of TCP, since this document only detects and quantifies reordering and does not change the TCP behavior. General security considerations for SACK based loss recovery are outlined in [RFC6675].

10. Acknowledgments

   The authors thank the flowgrind [Flowgrind] authors and contributors for their performance measurement tool, which give us a powerful tool to analyze TCP's congestion control and loss recovery behavior in detail.

11. References

11.1. Normative References

   [I-D.zimmermann-tcpm-reordering-reaction]
              Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann,
              "An adaptive Robustness of TCP to Non-Congestion Events",
              draft-zimmermann-tcpm-reordering-reaction-01 (work
              in progress), November 2013.

   [MM96]   Mathis, M. and J. Mahdavi, "Forward Acknowledgement:
              Refining TCP Congestion Control", ACM SIGCOMM 1996
              Proceedings, in ACM Computer Communication Review 26 (4),
              pp. 281-292, October 1996.

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

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

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

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

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

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

   [RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective
              Acknowledgement (DSACKs) and Stream Control Transmission
              Protocol (SCTP) Duplicate Transmission Sequence Numbers
              (TSNs) to Detect Spurious Retransmissions", RFC 3708,
              February 2004.

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

11.2. Informative References

   [BA02]   Blanton, E. and M. Allman, "On Making TCP More Robust to
              Packet Reordering", ACM Computer Communication
              Review vol.32, no. 1, pp. 20-30, January 2002.

   [BPS99]   Bennett, J., Partridge, C., and N. Shectman, "Packet

reordering is not pathological network behavior", IEEE/ACM
Transactions on Networking vol. 7, no. 6, pp. 789-798,
December 1999.

[BS02]   Bellardo, J. and S. Partridge, "Measuring Packet
            Reordering", Proceedings of the 2nd ACM SIGCOMM Workshop
            on Internet Measurment (IMW'02) pp. 97-105, November 2002.

[Flowgrind]
            "Flowgrind Home Page", <http://www.flowgrind.net>.

[GPL04]   Gharai, L., Perkins, C., and T. Lehman, "Packet
            Reordering, High Speed Networks and Transport Protocol
            Performance", Proceedings of the 13th IEEE International
            Conference on Computer Communications and Networks
            (ICCCN'04) pp. 73-78, October 2004.

[Hoe96]   Hoe, J., "Improving the Start-up Behavior of a Congestion
            Control Scheme for TCP", Proceedings of the Conference on
            Applications, Technologies, Architectures, and Protocols
            for Computer Communication (SIGCOMM'96) pp. 270-280,
            August 1996.

[IPPM]   "IP Performance Metrics (IPPM) Working Group",
            <http://www.ietf.org/html.charters/ippm-charter.html>.

[Jai+07]   Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and D.
            Towsley, "Measurement and Classification of Out-of-
            Sequence Packets in a Tier-1 IP Backbone", IEEE/ACM
            Transactions on Networking vol. 15, no. 1, pp. 54-66,
            February 2007.

[KP87]   Karn, P. and C. Partridge, "Improving Round-Trip Time
            Estimates in Reliable Transport Protocols", ACM SIGCOMM
            Computer Communication Review vol. 17, no. 5, pp. 2-7,
            November 1987.

[Linux]   "The Linux Project", <http://www.kernel.org>.

[Mat+97]   Mathis, M., Semke, J., Mahdavi, J., and T. Ott, "The
            Macroscopic Behavior of the TCP Congestion Avoidance
            Algorithm", ACM SIGCOMM Computer Communication Review vol.
            27, no. 3, pp. 67-82, July 1997.

[Mel+08]   Mellia, M., Meo, M., Muscariello, L., and D. Rossi,
            "Passive analysis of TCP anomalies", Computer
            Networks vol. 52, no. 14, pp. 2663-2676, October 2008.

   [Pax97]   Paxson, V., "End-to-End Internet Packet Dynamics", IEEE/
               ACM Transactions on Networking vol. 7, no.3, pp. 277-292,
               June 1997.

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

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

   [RFC2960]   Stewart, R., Xie, Q., Morneault, K., Sharp, C.,
               Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M.,
               Zhang, L., and V. Paxson, "Stream Control Transmission
               Protocol", RFC 2960, October 2000.

   [RFC4653]   Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton,
               "Improving the Robustness of TCP to Non-Congestion
               Events", RFC 4653, August 2006.

   [RFC4737]   Morton, A., Ciavattone, L., Ramachandran, G., Shalunov,
               S., and J. Perser, "Packet Reordering Metrics", RFC 4737,
               November 2006.

   [RFC5682]   Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata,
               "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting
               Spurious Retransmission Timeouts with TCP", RFC 5682,
               September 2009.

   [RFC6298]   Paxson, V., Allman, M., Chu, J., and M. Sargent,
               "Computing TCP's Retransmission Timer", RFC 6298,
               June 2011.

   [RFC6582]   Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The
               NewReno Modification to TCP's Fast Recovery Algorithm",
               RFC 6582, April 2012.

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

   [ZM04]   Zhou, X. and P. Mieghem, "Reordering of IP Packets in
               Internet", Lecture Notes in Computer Science vol. 3015,
               pp. 237-246, April 2004.

   [Zha+03]   Zhang, M., Karp, B., Floyd, S., and L. Peterson, "RR-TCP:
               A Reordering-Robust TCP with DSACK", Proceedings of the
               11th IEEE International Conference on Network Protocols

          (ICNP'03) pp. 95-106, November 2003.

Appendix A. Changes from previous versions of the draft

   This appendix should be removed by the RFC Editor before publishing
   this document as an RFC.

A.1. Changes from draft-zimmermann-tcpm-reordering-detection-00

   o Improved the wording throughout the document.
   o Replaced and updated some references.

Authors' Addresses

   Alexander Zimmermann
   NetApp, Inc.
   Sonnenallee 1
   Kirchheim 85551
   Germany

   Phone: +49 89 900594712
   Email: alexander.zimmermann@netapp.com

   Lennart Schulte
   Aalto University
   Otakaari 5 A
   Espoo 02150
   Finland

   Phone: +358 50 4355233
   Email: lennart.schulte@aalto.fi

   Carsten Wolff
   credativ GmbH
   Hohenzollernstrasse 133
   Moenchengladbach 41061
   Germany

   Phone: +49 2161 4643 182
   Email: carsten.wolff@credativ.de

   Arnd Hannemann
   credativ GmbH
   Hohenzollernstrasse 133
   Moenchengladbach 41061
   Germany

   Phone: +49 2161 4643 134
   Email: arnd.hannemann@credativ.de