TCP Maintenance and Minor Extensions
(TCPM) WG
Internet-Draft
Intended status: Experimental
Expires: May 22, 2014

A. Zimmermann
NetApp, Inc.
L. Schulte
Aalto University
C. Wolff
A. Hannemann
credativ GmbH

November 18, 2013

Making TCP Adaptively Robust to Non-Congestion Events
draft-zimmermann-tcpm-reordering-reaction-00

Abstract

This document specifies an adaptive Non-Congestion Robustness (aNCR) mechanism 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.

TCP-aNCR is designed to mitigate this performance degradation by adaptively increasing the number of duplicate acknowledgments required to trigger loss recovery, based on the current state of the connection, in an effort to better disambiguate true segment loss from segment reordering. This document specifies the changes to TCP and TCP-NCR (on which this specification build on) and discusses the costs and benefits of these modifications.

Status of this Memo

This Internet-Draft will expire on May 22, 2014.

Copyright Notice

Table of Contents

1. Introduction

One strength of the Transmission Control Protocol (TCP) [RFC0793] lies in its ability to adjust its sending rate according to the perceived congestion in the network [RFC5681]. In the absence of explicit notification of congestion from the network, TCP uses segment loss as an indication of congestion (i.e., assuming queue overflow). A TCP receiver sends cumulative acknowledgments (ACKs) indicating the next sequence number expected from the sender for arriving segments [RFC0793]. When segments arrive out of order, duplicate ACKs are generated. As specified in [RFC5681], a TCP sender uses the arrival of three duplicate ACKs as an indication of segment loss. The TCP sender retransmits the segment assumed lost and reduces the sending rate, based on the assumption that the loss was caused by resource contention on the path. The TCP sender does not assume loss on the first or second duplicate ACK, but waits for three duplicate ACKs to account for minor packet reordering. However, the use of this constant DupThresh leads to performance degradation if the extent of the packet reordering in the network increases [RFC4653].

Whenever interoperability with the TCP congestion control and loss recovery standard [RFC5681] is a prerequisite, increasing the duplicate acknowledgment threshold (DupThresh) is the method of choice to prevent any negative impact - in particular, a spurious Fast Retransmit and Fast Recovery phase - that packet reordering has on TCP. However, this procedure also delays a Fast Retransmit by increasing the DupThresh, and therefore has costs and risks, too. According to [Zha+03], these are: (1) a delayed response to congestion in the network, (2) a potential expiration of the retransmission timer, and (3) a significant increase in the end-to-end delay for lost segments.

In the current TCP standard, congestion control and loss recovery are tightly coupled: when the oldest outstanding segment is declared lost, a retransmission is triggered, and the sending rate is reduced on the assumption that the loss is due to resource contention [RFC5681]. Therefore, any change to DupThresh causes not only a change to the loss recovery, but also to the congestion control response. TCP-NCR [RFC4653] addresses this problem by defining two extensions to TCP’s Limited Transmit [RFC3042] scheme: Careful and Aggressive Extended Limited Transmit.

The first variant of the two, Careful Limited Transmit, sends one previously unsent segment in response to duplicate acknowledgments for every two segments that are known to have left the network. This effectively halves the sending rate, since normal TCP operation sends one new segment for every segment that has left the network.

Further, the halving starts immediately and is not delayed until a retransmission is triggered. In the case of packet reordering (i.e., not segment loss), TCP-NCR restores the congestion control state to its previous state after the event.

The second variant, Aggressive Limited Transmit, transmits one previously unsent data segment in response to duplicate acknowledgments for every segment known to have left the network. With this variant, while waiting to disambiguate the loss from a reordering event, ACK-clocked transmission continues at roughly the same rate as before the event started. Retransmission and the sending rate reduction happen per [RFC5681] [RFC6675], albeit after a delay caused by the increased DupThresh. Although this approach delays legitimate rate reductions (possibly slightly, and temporarily aggravating overall congestion on the network), the scheme has the advantage of not reducing the transmission rate in the face of packet reordering.

A basic requirement for preventing an avoidable expiration of the retransmission timer is to generally ensure that an increased DupThresh can potentially be reached in time so that Fast Retransmit is triggered and Fast Recovery is completed before the RTO expires. Simply increasing DupThresh before retransmitting a segment can make TCP brittle to packet or ACK loss, since such loss reduces the number of duplicate ACKs that will arrive at the sender from the receiver. For instance, if cwnd is 10 segments and one segment is lost, a DupThresh of 10 will never be met, because duplicate ACKs corresponding to at most 9 segments will arrive at the sender. To mitigate this issue, the TCP-NCR [RFC4653] modification makes two fundamental changes to the way [RFC5681] [RFC6675] currently operates.

First, as mentioned above, TCP-NCR [RFC4653] extends TCP's Limited Transmit [RFC3042] scheme to allow for the sending of new data segment while the TCP sender stays in the 'disorder' state while it is disambiguating loss and reordering. This new data serves to increase the likelihood that enough duplicate ACKs arrive at the sender to trigger loss recovery, if it is appropriate. Second, DupThresh is increased from the current fixed value of three [RFC5681] to approximately a until congestion window's worth of data has left the network. Since cwnd represents the amount of data a TCP sender can transmit in one round-trip time (RTT), this corresponds to approximately the largest amount of time a TCP sender can wait before the costly retransmission timeout may be triggered.

Of vital importance is that TCP-NCR [RFC4653] holds DupThresh not constant, but dynamically adjusts it on each SACK to the current amount of outstanding data, which depends not only on the congestion window, but also on the receiver's advertised window. In addition, available data from the application is all-important in generating a sufficient number of duplicate ACKs for reaching DupThresh and a transition to the 'recovery' state.

Regarding the problem of packet reordering, TCP-NCR's [RFC4653] decision of waiting to receive notice that cwnd bytes have left the network before deciding whether the root cause is loss or reordering is essentially a trade-off between making the best decision regarding the cause of the duplicate ACKs and responsiveness, and represents a good compromise between avoiding spurious fast retransmits and avoiding unnecessary RTOs. On the other hand, if there is no visible packet reordering on the network path - which today is the rule and not the exception - or the delay caused by the reordering is very low, delaying Fast Retransmit is unnecessary in the case of congestion, and data is delivered to the application up to one RTT later. Especially for delay-sensitive applications, such as a terminal session over SSH, this is generally undesirable. By dynamically adapting DupThresh not only to the amount of outstanding data but also to the perceived packet reordering on the network path, this issue can be offset. This is the key idea behind the TCP-aNCR algorithm.

This document specifies a set of TCP modifications to provide an adaptive Non-Congestion Robustness (aNCR) mechanism for TCP. The TCP-aNCR modifications lend themselves to incremental deployment. Only the TCP implementation on the sender side requires modification. The changes themselves are modest. TCP-aNCR is built on top of the TCP Selective Acknowledgments Option [RFC2018] and the SACK-based loss recovery scheme given in [RFC6675] and represents an enhancement of the original TCP-NCR mechanism [RFC4653]. Currently, TCP-aNCR is an independent approach of making TCP more robust to packet reordering. It is not clear if upcoming versions of this draft TCP-aNCR will obsolete TCP-NCR or not.

It should be noted that the TCP-aNCR algorithm in this document could be easily adapted to the Stream Control Transmission Protocol (SCTP) [RFC2960], since SCTP uses congestion control algorithms similar to TCP (and thus has the same reordering robustness issues).

The remainder of this document is organized as follows. Section 3 provides a high-level description of the TCP-aNCR mechanism. Section 4 defines TCP-aNCR's requirements for an appropriate detection and quantification algorithm. Section 5 specifies the TCP-aNCR algorithm and Section 6 discusses each step of the algorithm in detail. Section 7 provides a discussion of several design decisions behind TCP-aNCR. Section 8 discusses interoperability issues related to introducing TCP-aNCR. Finally, related work is presented in

Section 9 and security concerns in Section 11.

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 described in [RFC0793] (SND.NXT), [RFC5681] (cwnd, rwnd, ssthresh, FlightSize, IW), [RFC6675] (pipe, DupThresh, SACK scoreboard), and [RFC6582] (recover). 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.

The following specification depends on the standard TCP congestion control and loss recovery algorithms and the SACK-based loss recovery scheme given in [RFC5681], respectively [RFC6675]. The algorithm presents an enhancement of TCP-NCR [RFC4653]. The reader is assumed to be familiar with the algorithms specified in these documents.

3. Basic Concept

The general idea behind the TCP-aNCR algorithm is to extend the TCP-NCR algorithm [RFC4653], so that - based on an appropriate packet reordering detection and quantification algorithm (see Section 4) - TCP congestion control and loss recovery [RFC5681] is adaptively adjusted to the actual perceived packet reordering on the network path.

TCP-NCR [RFC4653] increases DupThresh from the current fixed value of three duplicate ACKs [RFC5681] to approximately until a congestion window of data has left the network. Since cwnd represents the amount of data a TCP sender can transmit in one RTT, the choice to trigger a retransmission only after a cwnd's worth of data is known to have left the network represents roughly the largest amount of time a TCP sender can wait before the RTO may be triggered. The approach chosen in TCP-aNCR is to take TCP-NCR's DupThresh as an upper bound for an adjustment of the DupThresh that is adaptive to the actual packet reordering on the network path.

Using TCP-NCR's DupThresh as an upper bound decouples the avoidance of spurious Fast Retransmits from the avoidance of unnecessary retransmission timeouts. Therefore, the adaptive adjustment of the DupThresh to current perceived packet reordering can be conducted without taking any retransmission timeout avoidance strategy into account. This independence allows TCP-aNCR to quickly respond to perceived packet reordering by setting its DupThresh so that it always corresponds to the minimum of the maximum possible (TCP-NCR's DupThresh) and the maximum measured reordering extent since the last RTO. The reordering extent used by TCP-aNCR is by itself not a static absolute reordering extent, but a relative reordering extent (see Section 4).

4. Appropriate Detection and Quantification Algorithms

If the TCP-aNCR algorithm is implemented at the TCP sender, it MUST be implemented together with an appropriate packet reordering detection and quantification algorithm that is specified in a standards track or experimental RFC.

Designers of reordering detection algorithms who want their algorithms to work together with the TCP-aNCR algorithm SHOULD reuse the variable "ReorExtR" (relative reordering extent) with the semantics and defined values specified in [I-D.zimmermann-tcpm-reordering-detection]. A ReorExtR given by the detection algorithm holds a value ranging from 0 to 1 which holds the new measured reordering sample as a fraction of the data in flight. TCP-aNCR then saves this new fraction if it is greater than the current value.

5. The TCP-aNCR Algorithm

When both the Nagle algorithm [RFC0896] [RFC1122] and the TCP Selective Acknowledgment Option [RFC2018] are enabled for a connection, a TCP sender MAY employ the following TCP-aNCR algorithm to dynamically adapt TCP’s congestion control and loss recovery [RFC5681] to the currently perceived packet reordering on the network path.

Without the Nagle algorithm, there is no straightforward way to accurately calculate the number of outstanding segments in the network (and, therefore, no good way to derive an appropriate DupThresh) without adding state to the TCP sender. A TCP connection that does not use the Nagle algorithm SHOULD NOT use TCP-aNCR. The adaptation of TCP-aNCR to an implementation that carefully tracks the sequence numbers transmitted in each segment is considered future work.

A necessary prerequisite for TCP-aNCR’s adaptability is that a TCP sender has enabled an appropriate detection and quantification algorithm that complies with the requirements defined in Section 4. If such an algorithm is either non-existent or not used, the behavior of TCP-aNCR is completely analogous to the TCP-NCR algorithm as defined in [RFC4653]. If a TCP sender does implement TCP-aNCR, the implementation MUST follow the various specifications provided in Sections 5.1 to 5.7.

5.1. Initialization during Connection Establishment

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

(C.1) Depending on which variant of Extended Limited Transmit should be executed, the constant LT_F MUST initialized as follows. For Careful Extended Limited Transmit:

LT_F = 2/3

For Aggressive Extended Limited Transmit:

LT_F = 1/2

This constant reflects the fraction of outstanding data (including data sent during Extended Limited Transmit) that must be SACKed before a retransmission is at the latest triggered.

(C.2) If TCP-aNCR should adaptively adjust the DupThresh to the current perceived packet reordering on the network path, then the variable ReorExtR, which stores the maximum relative reordering extent, MUST initialized as:

ReorExtR = 0

Otherwise the dynamically adaptation of TCP-aNCR SHOULD be disabled by setting

ReorExtR = -1

A relative reordering extent of 0 results in the standard DupThresh of three duplicate ACKs, as defined in [RFC5681]. A fixed relative reordering extent of -1 results in the TCP-NCR behavior from [RFC4653].

5.2. Initializing Extended Limited Transmit

If the SACK scoreboard is empty upon the receipt of a duplicate ACK (i.e., the TCP sender has received no SACK information from the receiver), a TCP sender MUST enter Extended Limited Transmit by initialize the following five state variables in the TCP Transmission Control Block:

(I.1) The TCP sender MUST save the current outstanding data:

FlightSizePrev = FlightSize

(I.2) The TCP sender MUST save the highest sequence number transmitted so far:

recover = SND.NXT - 1

Note: The state variable 'recover' from [RFC6582] can be reused, since NewReno TCP uses 'recover' at the initialization of a loss recovery procedure, whereas TCP-aNCR uses 'recover' *before* loss recovery.

(I.3) The TCP sender MUST initialize the variable 'skipped' that tracks the number of segments for which an ACK does not trigger a transmission during Careful Limited Transmit:

skipped = 0

During Aggressive Limited Transmit, 'skipped' is not used.

(I.4) The TCP sender MUST set DupThresh based on the current FlightSize:

DupThresh = max (LT_F * (FlightSize / SMSS), 3)

The lower bound of DupThresh = 3 is kept from [RFC5681]

[RFC6675].

(I.5) If (ReorExt != -1) holds, then the TCP sender MUST set DupThresh based on the relative reordering extent ReorExtR:

```
DupThresh = max (min (DupThresh,
                      ReorExtR * (FlightSize / SMSS)), 3)
```

In addition to the above steps, the incoming ACK MUST be processed with the (E) series of steps in Section 5.3.

5.3. Executing Extended Limited Transmit

On each ACK that a) arrives after TCP-aNCR has entered the Extended Limited Transmit phase (as outlined in Section 5.2) *and* b) carries new SACK information, *and* c) does *not* advance the cumulative ACK point, the TCP sender MUST use the following procedure.

(E.1) The TCP sender MUST update the SACK scoreboard and uses the SetPipe() procedure from [RFC6675] to set the 'pipe' variable (which represents the number of bytes still considered "in the network"). Note: the current value of DupThresh MUST be used by SetPipe() to produce an accurate assessment of the amount of data still considered in the network.

(E.2) The TCP sender MUST initialize the variable 'burst' that tracks the number of segments that can at most be sent per ACK to the size of the Initial Window (IW) [RFC5681]:

```
burst = IW
```

(E.3) If a) (cwnd - pipe - skipped >= 1 * SMSS) holds, *and* b) the receive window (rwnd) allows to send SMSS bytes of previously unsent data, *and* c) there are SMSS bytes of previously unsent data available for transmission, then the TCP sender MUST transmit one segment of SMSS bytes. Otherwise, the TCP sender MUST skip to step (E.7).

(E.4) The TCP sender MUST increment 'pipe' by SMSS bytes and MUST decrement 'burst' by SMSS bytes to reflect the newly transmitted segment:

```
pipe = pipe + SMSS
burst = burst - SMSS
```

(E.5) If Careful Limited Transmit is used, 'skipped' MUST be incremented by SMSS bytes to ensure that the next SMSS bytes of SACKed data processed do not trigger a Limited Transmit transmission.

```
skipped = skipped + SMSS
```

(E.6) If (burst > 0) holds, the TCP sender MUST return to step (E.3) to ensure that as many bytes as appropriate are transmitted. Otherwise, if more than IW bytes were SACKed by a single ACK, the TCP sender MUST skip to step (E.7). The additional amount of data becomes available again by the next received duplicate ACK and the re-execution of SetPipe().

(E.7) The TCP sender MUST save the maximum amount of data that is considered to have been in the network during the last RTT:

```
pipe_max = max (pipe, pipe_max)
```

(E.8) The TCP sender MUST set DupThresh based on the current FlightSize:

```
DupThresh = max (LT_F * (FlightSize / SMSS), 3)
```

The lower bound of DupThresh = 3 is kept from [RFC5681] [RFC6675].

(E.9) If (ReorExt != -1) holds, then the TCP sender MUST set DupThresh based on the relative reordering extent ReorExtR:

```
DupThresh = max (min (DupThresh,
                      ReorExtR * (FlightSize / SMSS)), 3)
```

5.4. Terminating Extended Limited Transmit

On the receipt of a duplicate ACK that a) arrives after TCP-aNCR has entered the Extended Limited Transmit phase (as outlined in Section 5.2) *and* b) advances the cumulative ACK point, the TCP sender MUST use the following procedure.

The arrival of an acceptable ACK that advances the cumulative ACK point while in Extended Limited Transmit, but before loss recovery is triggered, signals that a series of duplicate ACKs was caused by reordering and not congestion. Therefore, Extended Limited Transmit will be either terminated or re-entered.

(T.1) If the received ACK extends not only the cumulative ACK point, but *also* carries new SACK information (i.e., the ACK is both an acceptable ACK and a duplicate ACK), the TCP sender MUST restart Extended Limited Transmit and MUST go to step (T.2). Otherwise, the TCP sender MUST terminate it and MUST skip to step (T.3).

(T.2) If the Cumulative Acknowledgment field of the received ACK covers more than recover (i.e., SEG.ACK > recover), Extended Limited Transmit has transmitted one cwnd worth of data without any losses and the TCP sender MUST update the following state variables by

```
FlightSizePrev = pipe_max
pipe_max = 0
```

and MUST go to step (I.2) to re-start Extended Limited Transmit. Otherwise if (SEG.ACK <= recover) holds, the TCP sender MUST go to step (I.3). This ensures that in the event of a loss the cwnd reduction is based on a current value of FlightSizePrev.

The following steps are executed only if the received ACK does *not* carry SACK information. Extended Limited Transmit will be terminated.

(T.3) A TCP sender MUST set ssthresh to:

```
ssthresh = max (cwnd, ssthresh)
```

This step provides TCP-aNCR with a sense of "history". If the next step (T.4) reduces the congestion window, this step ensures that TCP-aNCR will slow-start back to the operating point that was in effect before Extended Limited Transmit.

(T.4) A TCP sender MUST reset cwnd to:

```
cwnd = FlightSize + SMSS
```

This step ensures that cwnd is not significantly larger than the amount of data outstanding, a situation that would cause a line rate burst.

(T.5) A TCP is now permitted to transmit previously unsent data as allowed by cwnd, FlightSize, application data availability, and the receiver’s advertised window.

5.5. Entering Loss Recovery

The receipt of an ACK that results in deeming the oldest outstanding segment is lost via the algorithms in [RFC6675] terminates Extended Limited Transmit and initializes the loss recovery according to [RFC6675]. One slight change to [RFC6675] MUST be made, however.

(Ret) In Section 5, step (4.2) of [RFC6675] MUST be changed to:

```
ssthresh = cwnd = (FlightSizePrev / 2)
```

This ensures that the congestion control modifications are made with respect to the amount of data in the network before FlightSize was increased by Extended Limited Transmit.

Once the algorithm in [RFC6675] takes over from Extended Limited Transmit, the DupThresh value MUST be held constant until the loss recovery phase terminates.

5.6. Reordering Extent

Whenever the additional detection and quantification algorithm (see Section 4) detects and quantifies a new reordering event, the TCP sender MUST update the state variable ReorExtR.

(Ext) Let ReorExtR_New the newly determined relative reordering extent:

```
ReorExtR = min (max (ReorExtR, ReorExtR_New), 1)
```

5.7. Retransmission Timeout

The expiration of the retransmission timer SHOULD be interpreted as an indication of a path characteristics change, and the TCP sender SHOULD reset DupThresh to the default value of three.

(RTO) If an RTO occurs, a TCP sender SHOULD reset ReorExtR:

```
ReorExtR = 0
```

6. Protocol Steps in Detail

Upon the receipt of the first duplicate ACK in the ’open’ state (the SACK scoreboard is empty), the TCP sender starts to execute TCP-aNCR by entering the ’disorder’ state and the initialization of Extended Limited Transmit. First, the TCP sender saves the currently amount of outstanding data as well as the highest sequence number transmitted so far (SND.NXT - 1) (steps (I.1) and (I.2)). In addition, if the TCP connection uses the careful variant of the Extended Careful Limited Transmit (step (C.1)), the 'skipped' variable, which tracks the number of segments for which an ACK does not trigger a transmission during Careful Limited Transmit, is initialized with zero (step (I.3)). The last step during the initialization is the determination of DupThresh. Depending on whether TCP-aNCR has been configured during the connection establishment to adaptively adjust to the currently perceived packet reordering on the path (step (C.2)), DupThresh is either determined exclusively based on the current FlightSize (as TCP-NCR [RFC4653] does) or, in addition, also based on the relative extent reordering (steps (I.4) and (I.5)).

Depending on which variant of Extended Limited Transmit should be executed, the constant LT_F must be set accordingly (step (C.1)). This constant reflects the fraction of outstanding data (including data sent during Extended Limited Transmit) that must be SACKed before a retransmission is triggered at the latest (which is the case when a DupThresh that is based on relative reordering extent is larger then TCP-NCR's DupThresh). Since Aggressive Limited Transmit sends a new segment for every segment known to have left the network, a total of approximately cwnd segments will be sent, and therefore ideally a total of approximately 2*cwnd segments will be outstanding when a retransmission is finally triggered. DupThresh is then set to LT_F = 1/2 of 2*cwnd (or about 1 RTT's worth of data) (see step (I.4)). The factor is different for Careful Limited Transmit, because the sender only transmits one new segment for every two segments that are SACKed and therefore will ideally have a total of maximum of 1.5*cwnd segments outstanding when the retransmission is triggered. Hence, the required threshold is LT_F=2/3 of 1.5*cwnd to delay the retransmission by roughly 1 RTT.

For each duplicate ACK received in the 'disorder' state, which is not an acceptable ACK, i.e., it carries new SACK information, but does not advance the cumulative ACK point, Extended Limited Transmit is executed. First, the SACK scoreboard is updated and based on the current value, DupThresh, the amount of outstanding data (step (E.1)). Furthermore, the state variable 'burst' that indicates the number of segments that can be sent at most for of each received ACK is initialized to the size of the initial window [RFC6928] (step E.2)). If more than IW bytes were SACKed by a single ACK, the additional amount of data becomes available again by the next received duplicate ACK and the re-execution of SetPipe() (step (E.1)).

Next, if new data is available for transmission and both the congestion window and the receiver window allow to send SMSS bytes of previously unsent data, a segment of SMSS bytes is sent (step (E.3)). Subsequently, the corresponding state variables 'pipe', 'burst' and - optionally - 'skipped' are updated (steps (E.4) and (E.5)). If, due to the current size of the congestion and receiver windows (step (E.2)), due to the current value of 'burst' (step (E.5)), no further segment may be sent, the processing of the ACK is terminated. Provided that the amount of data that is currently considered to be in the network is greater than the previously stored one, the new value is stored for later use (step (E.7)). Finally, to take into account the new data sent, DupThresh is updated (steps (E.6) and (E.7)).

The arrival of an acceptable ACK in the 'disorder' state that advances the cumulative ACK point during Extended Limited Transmit signals that a series of duplicate ACKs was caused by reordering and not congestion. Therefore, the receipt of an acceptable ACK that does not carry any SACK information terminates Extended Limited Transmit (step (T.1)). The slow start threshold is set to the maximum of its current value and the current value of cwnd (step (T.3)). Cwnd itself is set to the current value of FlightSize plus one segment (step (T.4)). As a result, the congestion window is not significantly larger than the current amount of outstanding data, so that a burst of data is effectively prevented. If new data is available for transmission and both the new values of cwnd and rwnd allow to send SMSS bytes of previously unsent data, a segment is send (step (T.5)).

On the other hand, if the received ACK is not only cumulative but at the same also SACKs new data - Extended Limited Transmit is not terminated but re-entered (step (T.1)). If the Cumulative Acknowledgment field of the received ACK covers more than 'recover', one cwnd worth of data has been transmitted during Extended Limited Transmit without any packet loss. Therefore, FlightSizePrev, the amount of outstanding data saved at the beginning of Extended Limited Transmit (step (I.1)), is considered outdated (step (T.2)). This step ensures that in the event of packet loss, the reduction of the cwnd is based on a up-to-date value, which reflects the number of bytes outstanding in the network (see Section 7). Finally, regardless of whether or not 'recover' is covered, Extended Limited Transmit is re-entered.

The second case that leads to a termination of Extended Limited Transmit is the receipt of an ACK that signals via the algorithm in [RFC6675] that the oldest outstanding segment is considered lost. If either DupThresh or more duplicate ACKs are received, or the oldest outstanding segment is deemed lost via the function IsLost() of [RFC6675], Extended Limited Transmit is terminated and SACK-based loss recovery is entered [RFC6675]. Once the algorithm in [RFC6675] takes over from Extended Limited Transmit, the DupThresh value MUST be held constant until loss recovery is terminated. The process of loss recovery itself is not changed by TCP-aNCR. The only exception is a slight change of the step (4.2) of RFC 6675 [RFC6675], which ensures that the adjustment made by the congestion control - halving the congestion window - is made with respect to the initial amount of outstanding data while Limited Transmit Extended is executed (step (Ret)). The use of FlightSize at this point would no longer be valid since the amount of outstanding data may double by executing Extended Limited Transmit.

7. Discussion of TCP-aNCR

The specification of TCP-aNCR represents an incremental update of RFC 4653 [RFC4653]. All changes made by TCP-aNCR can be divided into two categories. On one hand, they implement TCP-aNCR's ability to dynamically adapted TCP congestion control and loss recovery [RFC5681] to the currently perceived packet reordering on the network path. These include the use of a variable DupThresh and the use of a relative reordering extent. On the other hand, the changes that basically correct weaknesses of the original TCP-NCR algorithm and which are independent of TCP-aNCR adaptability. These include packet reordering during slow start, the prevention of bursts, and the persistent receipt of SACKs.

7.1. Variable Duplicate Acknowledgment Threshold

The central point of the TCP-aNCR algorithm is the usage of a DupThresh that is adaptable to the perceived packet reordering on the network path. Based on the actual amount of outstanding data, TCP-NCR's DupThresh represents roughly the largest amount of time a fast retransmit can safely be delayed before a costly retransmission timeout may be triggered. Therefore, to avoid an RTO, TCP-aNCR's packet-reordering-aware DupThresh is an upper bound of the one calculated in TCP-NCR (steps (I.5) and (E.9)). This decouples the avoidance of spurious Fast Retransmits from the avoidance of RTOs. It allows TCP-aNCR to react fast and efficiently to packet reordering, since the DupThresh always matches the minimum of the largest possible and largest detected reordering. With constant packet reordering in terms of the rate and delay, TCP-aNCR gives a DupThresh based on the relative reordering extent with an optimal delay for every bandwidth-delay-product. If TCP-aNCR should not adaptively adjust the DupThresh to the current perceived packet reordering on the network path (because for example an appropriate detection and quantification algorithm is not implemented), the dynamically adaptation of TCP-aNCR can be disabled, so that TCP-aNCR behaves like TCP-NCR [RFC4653].

7.2. Relative Reordering Extent

Whenever a new reordering event is detected and presented to TCP-aNCR in the form of a relative reordering extend ReorExtR, TCP-aNCR saves and uses the new ReorExtR if it is larger than the old one (step (EXT)). The upper bound of 1 assures that no excessively large value is used. A ReorExtR larger than one means that more than FlightSize bytes would have been received out-of-order before the reordered segment is received. The delay caused by the reordering is thus longer than the RTT of the TCP connection. Since the RTT is roughly the time a fast retransmit can safely be delayed before the retransmission has to be to avoid an RTO, a maximum ReorExtR of one seems to be a suitable value.

The expiration of the retransmission timer is interpreted by TCP-aNCR as an indication of a change in path characteristics, hence, the saved ReorExtR is assumed to be outdated and will be invalidated (step (RTO)). As a consequence, the relative reordering extent ReorExtR increases monotonically between two successive retransmission timeouts and corresponds to the maximum measured reordering extent since the last RTO. Other approaches would be an exponentially-weighted moving average (EWMA) or a histogram of the last n reordering extents. The main drawback of an EWMA is however that on average half of the detected reordering events would be larger than the saved reordering extend. Thus, only half of the spurious retransmits could be avoided. Applying an histogram could largely avoid the disadvantages of an EWMA, however, it would result in a not acceptable increase in memory usage.

In combination with the invalidation after an RTO, the advantage of using maximum is the low complexity as well as its fast convergence to the actual maximum reordering on the network path. As a result, the negative impact that packet reordering has on TCP’s congestion control and loss recovery can be avoided. A disadvantage of using a maximum is that if the delay caused by the reordering decreases over the lifetime of the TCP connection, a Fast Retransmit is unnecessarily long delayed. Nevertheless, since the negative impact reordering has on TCP’s congestion control and loss recovery is more substantial than the disadvantage of a longer delay, a decrease of the ReorExtR between RTOs is considered inappropriate.

7.3. Reordering during Slow Start

The arrival of an acceptable ACK during Extended Limited Transmit signals that previously received duplicate ACKs are the result of packet reordering and not congestion, so that Extended Limited Transmit is completed accordingly. Upon the termination of Extended Limited Transmit, and especially when using the Careful variant, TCP-

NCR (as well as TCP-aNCR) may be in a situation where the entire cwnd is not being utilized. Therefore, to mitigate a potential burst of segments, in step (T.2) TCP-NCR sets the slow start threshold to the FlightSize that was saved at the beginning of Extended Limited Transmit [RFC4653]. This step should ensure that TCP-NCR slow starts back to the operating point in use before Extended Limited Transmit.

Unfortunately, the assignment in step (T.2) is only correct if the TCP sender already was in congestion avoidance at the time Extended Limited Transmit was entered. Otherwise, if the TCP sender was instead in slow start, the value of ssthresh is greater than the saved FlightSize so that slow start prematurely concludes. This behavior can leave much of the network resources idle, and a long time may needed in order to use the full capacity. To mitigate this issue, TCP-aNCR sets the slow start threshold to the maximum of its current value and the current cwnd (step (T.3)). This continues slow start after a reordering event happening during slow start.

7.4. Preventing Bursts

In cases where a new single SACK covers more than one segment - this can happen either due to packet loss or packet reordering on the ACK path - TCP-NCR [RFC4653] sends an undesirable burst of data. TCP-aNCR solves this problem by limiting the burst size - the maximum of data that can send in response to a single SACK - to the Initial Window [RFC5681] while executing Extended Limited Transmit (steps (E.2), (E.4), and (E.6)). Since IW represents the amount of data that a TCP sender is able to send into the network safely without knowing its characteristics, it is a reasonable value for the burst size, too. If more than IW bytes were SACKed by a single ACK, the additional amount of data becomes available again by the next received duplicate ACK. Thus, the transmission of new segments is spread over the next received ACKs, so that micro bursts - a characteristic of packet reordering in the reverse path - are largely compensated.

Another situation that causes undesired bursts of segments with TCP-NCR is the receipt of an acceptable ACK during Careful Extended Limited Transmit. If multiple segments from a single window of data are delayed by packet reordering, typically the first acceptable ACK after entering the 'disorder' state acknowledges data not only cumulatively but also selectively. Hence, Extended Limited Transmit is not terminated but re-started. If the segments are delayed by the reordering for almost one RTT, then the amount of outstanding data in the network ('pipe') is approximately half the amount of data saved at the beginning of Extended Limited Transmit (FlightSizePrev). If the sequence numbers of the delayed segments are close to each other in the sequence number space, the acceptable ACK acknowledges only a small amount of data, so that FlightSize is still large. As a result, TCP-NCR sets the cwnd to FlightSizePrev in step (T.1). Since 'pipe' is only half of FlightSizePrev due to Careful Extended Limited Transmit, TCP-NCR sends a burst of almost half a cwnd worth of data in the subsequent step (T.3).

Note: Even in the case the sequence numbers of the delayed segments are not close to each other in the sequence number space and cwnd is set in step (T.1) to FlightSize + SMSS, a burst of data will emerge due to re-entering Extended Limited Transmit, because TCP-NCR sets 'skipped' to zero in step (I.2) and uses FlightSizePrev in step (E.2).

TCP-aNCR prevents such a burst by making a clear differentiation between terminating Extended Limited Transmit and a restarting Extended Limited Transmit (step T.1). Only the first case causes the congestion window to be set to the current FlightSize plus one segment. In the latter case, when re-entering Extended Limited Transmit, the congestion window is not adjusted and the original (T.1) of the TCP-NCR specification is omitted. The transmission of new data is then only performed after re-entering Extended Limited Transmit in step (E.2) of the TCP-aNCR specification, where the actual burst mitigation takes place.

7.5. Persistent receiving of Selective Acknowledgments

In some inconvenient cases it could happen that a TCP sender persistently receives SACK information due to reordering on the network path, e.g., if the segments are often and/or lengthily delayed by the packet reordering. With TCP-NCR, the persistent reception of SACKs causes Extended Limited Transmit to be entered with the first received duplicate ACK but never to be terminated if no packet loss occurs - for every received ACK, TCP-NCR either follows steps (E.1) to (E.6) or steps (T.1) to (T.4). In particular, TCP-NCR executes a) for every acceptable ACK step (T.4) and b) at any time step (I.1) again. Hence, the amount of outstanding data saved at the beginning of Extended Limited Transmit, FlightSizePrev, is never updated.

An emerging problem in this context is that during Extended Limited Transmit TCP-NCR determines the transmission of new segments in step (E.2) solely on the basis of FlightSizePrev, so that an interim increase of the cwnd is not considered (according to [RFC5681], the congestion window is increased for every received acceptable ACK that advances the cumulative ACK point, no matter if it carries SACK information or not). As a result, TCP-NCR can only very slowly determine the available capacity of the communication path.

TCP-aNCR addresses this problem by limiting the amount of data that is allowed to be sent into the network during Extended Limited Transmit not on the basis of FlightSizePrev, but on the size of the congestion window. The equation in step E.3 of the TCP-aNCR specification is therefore equal to the one used in [RFC6675] (except for the 'skipped' variable). If an acceptable ACK is received during the execution of Extended Limited Transmit, re-entering Extended Limited Transmit makes any increase in cwnd immediately available. Hence, even in the case when persistently receiving SACKs, the available capacity of the communication path can be determined quickly.

Another problem resulting from persistently receiving SACKs, and which is related to the increase in cwnd in response to received acceptable ACKs, is the reduction of cwnd due to a packet loss. When a packet is considered lost, the congestion control adjustment is done with respect to the amount of outstanding data at the beginning of Extended Limited Transmit, FlightSizePrev (step (Ret)). As in the previous case, an increase in cwnd is again not taken into account. A simple solution to the problem would be to perform the window reduction not on the basis of FlightSizePrev but analogous to step (E.2) based on the current size of cwnd.

A problem with this solution is that cwnd can potentially be increased, although the TCP connection is limited by the application and not by cwnd. Although [RFC2861] specifies that an increase of cwnd is only applicable if cwnd is fully utilized, this behavior is not specified by any standards track document. But even this conservative increase behavior is guaranteed to not be conservative enough. If, from a single window of data, both segments are delayed but also lost, cwnd would first be increased in response to each received acceptable ACKs, while subsequently reduced due to the lost segments, which would not result in a halving of the cwnd any more.

The solution proposed by TCP-aNCR reuses the state variable 'recover' from [RFC6582] and adapts the approach taken by NewReno TCP and SACK TCP to detect, with help of the state variable, the end of one loss recovery phase properly, allowing to recover multiple losses from a single window of data efficiently. Therefore, by entering the 'disorder' state and the starting Extended Limited Transmit, TCP-aNCR saves the highest sequence number sent so far in 'recover'. If a received acceptable ACK covers more than 'recover', one cwnd's worth of data has been transmitted during Extended Limited Transmit without any packet loss. Hence, FlightSizePrev can be updated by pipe_max, which reflects the maximum amount of data that is considered to have been in the network during the last RTT. This update takes an interim increase in cwnd into account, so that in case of packet loss, the reduction in cwnd can be based on the current value of

FlightSizePrev.

8. Interoperability Issues

TCP-aNCR requires that both the TCP Selective Acknowledgment Option [RFC2018] as well as a SACK-based loss recovery scheme compatible to one given in [RFC6675] are used by the TCP sender. Hence, compatibility to both specifications is REQUIRED.

8.1. Early Retransmit

The specification of TCP-aNCR in this document and the Early Retransmit algorithm specified in [RFC5827] define orthogonal methods to modify DupThresh. Early Retransmit allows the TCP sender to reduce the number of duplicate ACKs required to trigger a Fast Retransmit below the standard DupThresh of three, if FlightSize is less than 4*SMSS and no new segment can be sent. In contrast, TCP-aNCR allows, starting from the minimum of three duplicate ACKs, to increase the DupThresh beyond the standard of three duplicate ACKs to make TCP more robust to packet reordering, if the amount of outstanding data is sufficient to reach the increased DupThresh to trigger Fast Retransmit and Fast Recovery.

8.2. Congestion Window Validation

The increase of the congestion window during application-limited periods can lead to an invalidation of the congestion window, in that it no longer reflects current information about the state of the network, if the congestion window might never have been fully utilized during the last RTT. According to [RFC2861], the congestion window should, first, only be increased during slow-start or congestion avoidance if the cwnd has been fully utilized by the TCP sender and, second, gradually be reduced during each RTT in which the cwnd was not fully used.

A problem that arises in this context is that during Careful Extended Limited Transmit, cwnd is not fully utilized due to the variable 'skipped' (see step (E.3)), so that - strictly following [RFC2861] - the congestion window should not be increased upon the receipt of an acceptable ACK. A trivial solution of this problem is to include the variable 'skipped' in the calculation of [RFC2861] to determine whether the congestion window is fully utilized or not.

8.3. Reactive Response to Packet Reordering

As a proactive scheme with the aim to a priori prevent the negative impact that packet reordering has on TCP, TCP-aNCR can conceptually be combined with any reactive response to packet reordering, which attempts to mitigate the negative effects of reordering a posteriori. This is because the modifications of TCP-aNCR to the standard TCP congestion control and loss recovery [RFC6675] are implemented in the 'disorder' state and are performed by the TCP sender before it enters loss recovery, while reactive responses to packet reordering operate generally after entering loss recovery, by undoing the unnecessarily changes to the congestion control state.

If unnecessary changes to the congestion control state are undone after loss recovery, which is typically the case if a spurious Fast Retransmit is detected based on the DSACK option [RFC3708][RFC4015], since first ACK carrying a DSACK option usually arrives at a TCP sender only after loss recovery has already terminated, it might happen that the restoring of the original value of the congestion window is done at a time at which the TCP sender is already back in again in the disorder state and executing Extended Limited Transmit. While this is basically compatible with the TCP-aNCR specification - the undo simply represents an increase of the congestion window - however, some care must be taken that the combination of the algorithms does not lead to unwanted behavior.

8.4. Buffer Auto-Tuning

Although all modifications of the TCP-aNCR algorithm are implemented in the TCP sender, the receiver also potentially has a part to play. If some segments from a single window of data are delayed by the packet reordering in the network, all segments that are received in out-of-order have to be queued in the receive buffer until the holes in sequence space have been closed and the data can be delivered to the receiving application. In the worst case, which occurs if the TCP sender uses Aggressive Limited Transmit and the reordering delay is close to the RTT, TCP-aNCR increases the receiver's buffering requirement by up to an extra cwnd. Therefore, to maximize the benefits from TCP-aNCR, receivers should advertise a large window - ideally by using buffer auto-tuning algorithms - to absorb the extra out-of-order data. In the case that the additional buffer requirements are not met, the use of the above algorithm takes into account the reduced advertised window - with a corresponding loss in robustness to packet reordering.

9. Related Work

Over the past few years, several solutions have been proposed to improve the performance of TCP in the face of packet reordering. These schemes generally fall into one of two categories (with some overlap): mechanisms that try to prevent spurious retransmits from happening (proactive schemes) and mechanisms that try to detect spurious retransmits and undo the needless congestion control state changes that have been taken (reactive schemes).

[I-D.blanton-tcp-reordering], [Zha+03] and [LM05] attempt to prevent packet reordering from triggering spurious retransmits by using various algorithms to approximate the DupThresh required to disambiguate loss and reordering over a given network path at a given time. This basic principle is also used in TCP-aNCR. While [I-D.blanton-tcp-reordering] describes four basic approaches on how to increase the DupThresh and discusses pros and cons of these approaches, presents [Zha+03] a relatively complex algorithm that saves the reordering extents in a histogram and calculates the DupThresh in a way that a certain percentage of samples is smaller then the DupThresh. [LM05] uses an EWMA for the same purpose. Both algorithms do not prevent all the spurious retransmissions by design.

In contrast to the above mentioned algorithms Linux [Linux] implements a proactive scheme by setting the DupThresh to the highest detected reordering and resets only upon an RTO. To avoid a costly retransmission timeout due to the increased DupThresh Linux implements first an extension of the Limited Transmit algorithm, second limits the DupThresh to an upper bound of 127 duplicate ACKs, and third prematurely enters loss recovery if too few segments are in-flight to reach the DupThresh and no additional segments can send. Especially the last change is commendable since, besides TCP-NCR, none of the described algorithms in this section mention a similar concern.

[Boh+06] and [Bha+04] presents proactive schemes based on timers by which the DupThresh is ignored altogether. After the timer is expired TCP initialize the loss recovery. In [Bha+04] this timer has a length of one RTT and is started when the first duplicate ACK is received, whereas the approach taken in [Boh+06] solely relies on timers to detect packet loss without taking into account any other congestion signals such as duplicate ACKs. It assigns each segment send a timestamp and retransmits the segment if the corresponding timer fires.

TCP-NCR [RFC4653] tries to prevent spurious retransmits similar to [I-D.blanton-tcp-reordering] or [Zha+03] as it delays a retransmission to disambiguate loss and reordering. However, TCP-NCR takes a simplified approach by simply delay a retransmission by an amount based on the current cwnd (in comparison to standard TCP), while the other schemes use relatively complex algorithms in an attempt to derive a more precise value for DupThresh that depends on the current patterns of packet reordering. Many of the features offered by TCP-NCR have been taken into account while designing TCP-aNCR.

Besides the proactive schemes, several other schemes have been developed to detect and mitigate needless retransmissions after the fact. The Eifel detection algorithm [RFC3522], the detection based on DSACKs [RFC3708], and F-RTO scheme [RFC5682] represent approaches to detect spurious retransmissions, while the Eifel response algorithm [RFC4015], [I-D.blanton-tcp-reordering], and Linux [Linux] present respectively implement algorithms to mitigate the changes these events made to the congestion control state. As discussed in Section 8.3 TCP-aNCR could be used in conjunction with these algorithms, with TCP-aNCR attempting to prevent spurious retransmits and some other scheme kicking in if the prevention failed.

10. IANA Considerations

This memo includes no request to IANA.

11. Security Considerations

By taking dedicated actions so that the perceived packet reordering in the network is either underestimating or overestimating by the use of an relative and absolute reordering, an attacker or misbehaving TCP receiver has in regards to TCP's congestion control two options to bias a TCP-aNCR sender. An underestimation of the present packet reordering in the network occursi, if for example, a misbehaving TCP receiver already acknowledges segments while they are actually still in-flight, causing holes premature are closed in the sequence space of the SACK scoreboard. With regard to TCP-aNCR the result of an underestimated packet reordering is a too small DupThresh, resulting in a premature loss recovery execution. In context of TCP's congestion control the effects of such attacks are limited since the lower bound of TCP-aNCR's DupThresh is the default value of three duplicate ACKs [RFC5681], so that in worst case TCP-aNCR behaves equal to TCP SACK [RFC6675].

In contrast to an underestimation, an overestimation of the packet reordering in the network occurs, if for example, a misbehaving TCP receiver still further send SACKs for subsequent segments before it sends an acceptable ACK for the actually already received delayed segment, so that the hole in the sequence space of the SACK scoreboard is later closed. In the context of TCP-aNCR the result of such an overestimation is a too large DupThresh, so that in the case of a packet loss TCP's Loss Recovery is executed later than necessary. Similar to the previous case, the effects of delayed entry into the loss recovery are limited because on the one hand TCP-

NCR’s DupThresh is used as an upper bound for TCP-aNCR’s variable DupThresh so that the entrance to the loss recovery and the adaptation of the congestion window may be delayed at most one RTT. On the other hand, such a limited delay of the congestion control adjustment has even in the worst case only a limited impact on the performance of TCP connection and has generally been regarded as safe for use on the Internet [Ban+01].

12. Acknowledgments

The authors would like to thank Daniel Slot for his TCP-NCR implementation in Linux. We also thank the flowgrind [Flowgrind] authors and contributors for here performance measurement tool, which give us a powerful tool to analyze TCP’s congestion control and loss recovery behavior in detail.

13. References

13.1. Normative References

[I-D.zimmermann-tcpm-reordering-detection]
Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-00 (work in progress), November 2013.

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

[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.

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP’s Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

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

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

[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.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, April 2013.

13.2. Informative References

[Ban+01] Bansal, D., Balakrishnan, H., Floyd, S., and S. Shenker, "Dynamic Behavior of Slowly Responsive Congestion Control Algorithms", Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01) pp. 263-274, September 2001.

[Bha+04] Bhandarkar, S., Sadry, N., Reddy, A., and N. Vaidya, "TCP-DCR: A Novel Protocol for Tolerating Wireless Channel Errors", IEEE Transactions on Mobile Computing vol. 4, no. 5., pp. 517-529, September 2005.

[Boh+06] Bohacek, S., Hespanha, J., Lee, J., Lim, C., and K. Obraczka, "A New TCP for Persistent Packet Reordering", IEEE/ACM Transactions on Networking vol. 2, no. 14, pp. 369-382, April 2006.

[Flowgrind] "Flowgrind Home Page", <https://github.com/flowgrind/flowgrind>.

[I-D.blanton-tcp-reordering] Blanton, E., Dimond, R., and M. Allman, "Practices for TCP Senders in the Face of Segment Reordering", draft-blanton-tcp-reordering-00 (work in progress), February 2003.

[LM05] Leung, C. and C. Ma, "Enhancing TCP Performance to Persistent Packet Reordering", KICS Journal of Communications and Networks vol. 7, no. 3, pp. 385-393, September 2005.

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

[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.

[RFC2861] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", RFC 2861, June 2000.

[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.

[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.

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, February 2005.

[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.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, May 2010.

[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.

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