DCLOR: De-correlated Loss Recovery using SACK option for spurious timeouts.

Status of this Memo

Abstract

   A spurious timeout in TCP forces the sender to unnecessarily
   retransmit one complete congestion window of data into the network.
   In addition, TCP uses the rate of arrival of ACKs as the basic
   criterion for congestion control. TCP makes the assumption that the
   rate at which ACKs are received reflects the end-to-end state of the
   network in terms of congestion. But after a spurious-timeout, the
   ACKs don't reflect the end-to-end congestion state of the network,
   but only a part of it. In these cases, the slow-start behavior after
   a timeout can further add to network congestion. In this draft we
   propose changes to the TCP sender (no change is needed for TCP
   receiver) that can be used to solve the problem of both redundant-
   retransmission and network congestion after a spurious timeout.

---

1. Introduction

The response of a TCP sender after a retransmission timeout is
governed by the underlying assumption that a mid-stream timeout can
occur only if there is heavy congestion--manifested as packet
loss--in the network. Even though loss is often caused by congestion,
the loss recovery algorithm itself should only answer the question of
"what" data (i.e., what sequence number of data) to send. While on
the other hand, the congestion control algorithm should answer the
question of "how much" data to send. But after a timeout, TCP
addresses the issues of loss recovery and congestion control using a
single mechanism--send one segment per round trip timeout (RTO)
(answers the "how much" question) until an acknowledgment is
received. The single segment sent is always the first unacknowledged
outstanding packet in the retransmission queue (answers the "what"
question). Since the present TCP's loss recovery and congestion
control algorithms are coupled together, we call this "Correlated
Loss Recovery (CLOR)."

Although the assumption that a timeout can occur only if there is
severe congestion is valid for traditional wire-line networks, it
does not hold good for some other types of networks--networks where
packets can be stalled "in the network" for a significant duration
without being discarded. Typical examples of such networks are
cellular networks. In cellular networks, the link layer can
experience a relatively long disruption due to errors, and the link
layer protocol can keep these packets-in-error buffered as long as
the link layer disruption lasts.

In this document we present an alternative approach to loss recovery
and congestion control that "De-Correlates" Loss Recovery from
congestion congestion and allows independent choice on using a
particular TCP sequence number without compromising on the congestion
control principles of [RFC2581][RFC2914][RFC2861].
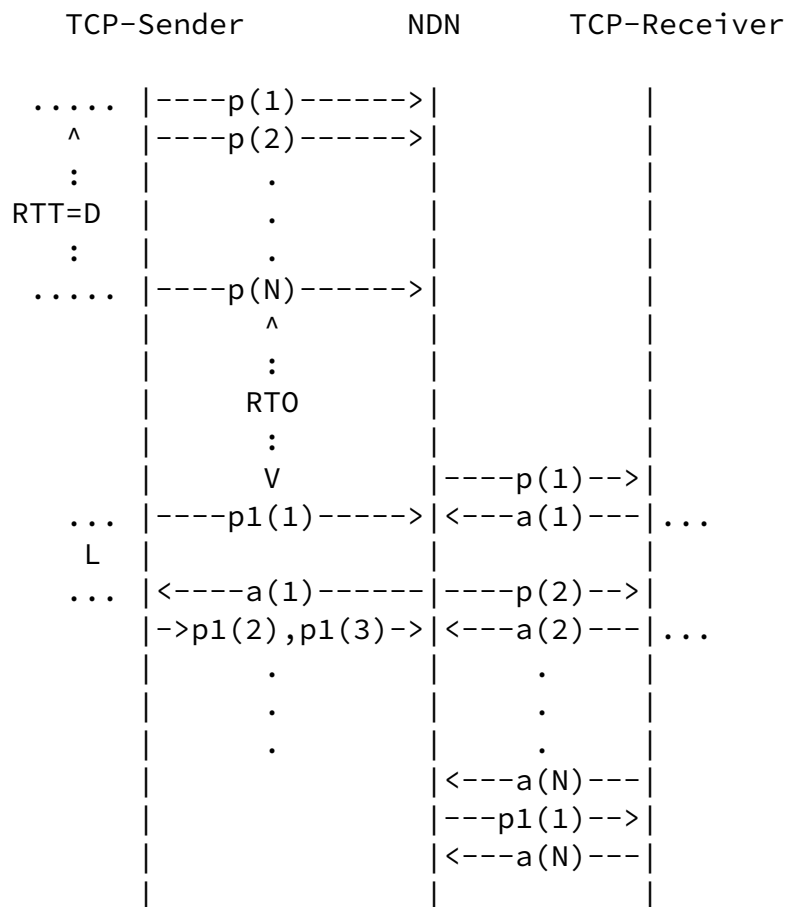
Although several drafts [LM02][LG03][SK03][BA02] have been presented
on this topic, we believe that none of them fully considers all the
problems associated with spurious timeouts. In the following section
we first describe these problems in more detail and then describe the
DCLOR mechanism in section-3.

2. Problem Description.

Let us assume that a TCP sender has sent N packets, p(1) ... p(N),
into the network and it's waiting for the ACK of p(1) (Figure-1). Due
to bad network conditions or some other problem, these packets are
excessively delayed at some some intermediary node NDN. Unlike
standard IP routers, the NDN keeps these packets buffered for a

---

relatively long period of time until these packets are forwarded to
their intended recipient. This excessive delay forces the TCP sender
to timeout and enter slow start.

Figure-1

```
       TCP-Sender              NDN        TCP-Receiver

   .....  |----p(1)------>|              |
     ^    |----p(2)------>|              |
     :    |       .       |              |
  RTT=D   |       .       |              |
     :    |       .       |              |
   .....  |----p(N)------>|              |
          |       ^       |              |
          |       :       |              |
          |      RTO       |              |
          |       :       |              |
          |       V       |----p(1)-->|
    ...   |----p1(1)----->|<---a(1)---|...
     L    |               |           |
    ...   |<----a(1)------|----p(2)-->|
          |->p1(2),p1(3)->|<---a(2)---|...
          |       .       |      .      |
          |       .       |      .      |
          |       .       |      .      |
          |               |<---a(N)---|
          |               |---p1(1)-->|
          |               |<---a(N)---|
          |               |           |
```

As far as the sender is concerned, a timeout is always interpreted as
heavy congestion. The TCP sender therefore makes the assumption that
all packets between p(1) and p(N) were lost in the network. To
recover from this misconstrued loss, the TCP sender retransmits P1(1)

(Px(k) represents the xth retransmission of packet with sequence
number k), and waits for the ACK a(1).

After some period of time when the network conditions at NDN improve,
the queued in packets are finally dispatched to their intended
recipient; in response the TCP receiver generates the ACK a(1). When
the TCP sender receives a(1), it's fooled into believing that a(1)
was generated in response to the retransmitted packet p1(1), while in
reality a(1) was generated in response to the originally transmitted
packet p(1). When the sender receives a(1), it increases its
congestion window to two, and retransmits p1(2) and p1(3). As the
sender receives more acknowledgments, it continues with

retransmissions and finally starts sending new data.

The following two sub sections examine the problems associated with
the above-mentioned TCP behavior.

2.1 Redundant Data Retransmission

The obvious and relatively easy-to-solve inefficiency of the above
algorithm is that the entire congestion window worth of data is
unnecessarily retransmitted. Although such retransmissions are
harmless to high-bandwidth, well-provisioned, backbone links (so long
they are infrequent), it could severely degrade the performance of
slow links.

In cases where bandwidth is a commodity at a premium, (e.g., cellular
networks), unnecessary retransmission can also be costly.

2.2 Congestion after Spurious Timeout

To analyze network congestion after spurious timeout, we compute the
worst case scenario packet loss in the system--assuming only TCP
connections to be present.

After the spurious timeout, the TCP sender sets its SS_THRESH to N/2.
Therefore, for the first N/2 ACKs received (i.e., ACK a(1) to a(N/2)
), the TCP sender will grow its congestion window by one and reach
the SS_THRESH value of N/2. For each ACK received, the TCP sender
sends 2 packets. Therefore, by the end of the slow start, the TCP
sender would have sent 2*(N/2) packets into the network. For the

remaining N/2 ACKs (i.e., ACKs between a(N/2+1) to a(N)) the TCP
sender will remain in the congestion avoidance phase and send one
packet for each ACK received--sending N/2 more data segments. The net
amount of data sent is therefore N/2 + N = 3N/2.

Please note that the entire 3N/2 packets are injected into the
network within a time period less than or equal to RTT in most cases.
The number of data segments that left the network during this time is
only N. Therefore, N/2 packets out of 3N/2 packets will be lost with
a very high probability. These N/2 lost packets, however, need not
come from the same connection, and such a data-burst will
unnecessarily penalize all the competing TCP connections that share
the same bottleneck router.

Going further ahead, let us assume there are M competing TCP
connections that share the same bottleneck router(s) with
C(0)(Figure-2). During the period of time while C(0) is stalled, the
TCP sender of C(0) does not use its network resources--the buffer
space--on the bottleneck router(s). The competing connections,
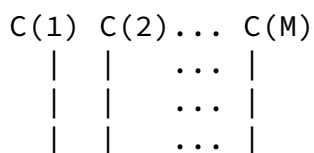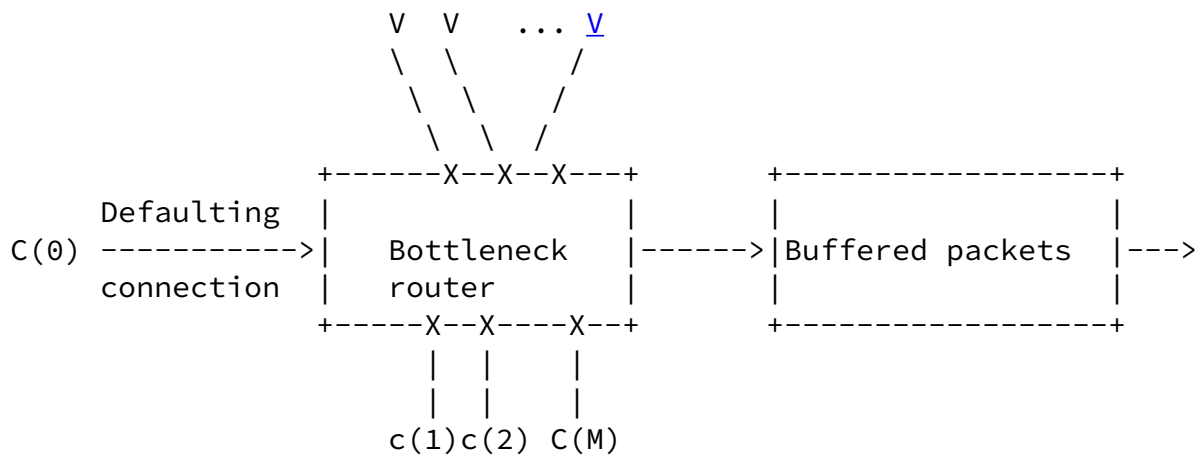
C(1)... C(M), however see this lack of activity as resource
availability and start growing their window by at least one segment
per RTT during this time period (by virtue of linear window increase
during congestion avoidance phase). For simplicity reasons, we
assume that each of these connections has the same round trip time of
RTT, and the idle time for C(0) is k*RTT (where k > RTO/RTT). Under
these assumptions, each of these competing connections will increase
their congestion window by k segments. Therefore the amount of
packets lost in the network due to slow start can be as high as:

$$N/2 + M*k \qquad ... (4)$$

the first term in the above equation is the packet loss due to slow
start, while the second term is the loss due to window growth of
completing connections (if the competing connections were in slow
start the response could have been worse).

Figure-2
                    C(1) C(2)... C(M)
                     | | ... |
                     | | ... |
                     | | ... |

```
                    V   V    ...  V
                     \   \      /
                      \   \    /
                       \   \  /
           +------X--X--X---+      +-----------------+
  Defaulting |              |      |                 |
  C(0) ----------->|  Bottleneck  |------>|Buffered packets  |--->
  connection |    router    |      |                 |
           +-----X--X----X--+      +-----------------+
                  |  |    |
                  |  |    |
              c(1)c(2) C(M)
```

   Based on the above equation, we note that the congestion state of the
   network depends upon the duration of spurious timeout. In our reponse
   algorithm we therefore take the time duration of spurious timeout
   into account reduce the data rate by half every RTO. Please note that
   this scheme works well only when the number of competing connections
   M does not vary too much while C(0) was stalled. A more conservative
   response algorithm should reduce the data rate to INIT_WINDOW if M is
   not bounded.

   In the following sections we describe an algorithm that solves the
   problem of both redundant retransmission and packet loss after a
   spurious timeout.

3. De-correlated Loss Recovery (DCLOR)

   The basic idea behind DCLOR is to send a new data segment from
   outside the sender's retransmission queue and wait for the ACK or
   SACK of the new data before initiating the response algorithm. Unlike
   slow-start where the response algorithm starts immediately after
   receiving the first ACK, DCLOR waits for the ACK/SACK of the new data
   sent after timeout before initiating loss recovery. The SACK block
   for new data contains sufficient information to determine all the
   packets that were lost into the network. Once the sequence number of
   lost packets is determined, the TCP sender grows its congestion
   window as determined by the SS_THRESH and it's congestion window.

3.1 Probe phase after a timeout

The following steps describe the response of a TCP sender on a
timeout:

   1. If the timeout occurs before the 3 way handshake is complete,
       the TCP sender's behavior is unchanged,

   2. After each timeout, the TCP sender MUST set its congestion
      window to:

$$cwnd = max(cwnd >> 1, IINIT\_WINDOW).$$

      The value of SS_THRESH MUST be left UNCHANGED at this point. The
      TCP sender should also count the number of packets in flight at
      this time, and keep it in a state variable stale_outstanding.

   3. The TCP sender SHOULD also reset all the SACK tag bits in its
      retransmission queue if this the first timeout.

   4. Instead of sending the first unacknowledged packet P1
      after a timeout, the TCP sender should *disregard* its
      congestion window and send ONE NEW MSS size data Pn+1.

      The TCP sender should also store the sequence number of the new
      segment in a new state variable called SS_PTR (for slow start
      pointer).

      If the sender does not have any new data outside its
      retransmission queue, or if the receiver's flow control window
      cannot sustain any new data, the TCP sender SHOULD send the
      highest sequence numbered MSS sized data chunk from its
      retransmission queue (i.e., it should send the last packet from
      its retransmission queue).

   5. A TCP sender MUST repeat step-2 to step-4 until it
      enters the Timeout-Recovery state as described in step 6.

3.2 Congestion Control After the probe phase

   6. For each ACK received with the ACK-sequence number
      less than SS_PTR, regardless of the value of the SS_THRESH, the
      TCP sender SHOULD NOT grow it's congestion window. If the ACK

contains a new SACK block, the SACK tag SHOULD be set in the
corresponding data packet. If new segments were ACKed, and the
congestion window allows, the TCP sender SHOULD send new data.
(Note: the idea here is that the congestion window should not be
grown in response to stale ACKs since these ACKs don't reflect
the end to end state of the network).

In addition, the TCP sender SHOULD NOT take any timer sample for
the stale ACKs. (NOTE: We do not attempt to change the RTT
calculation in an ad-hoc manner; we believe that this is a
reaseach problem that needs better network modelling before an
appropriate timer calculation can be found)

7. Step-6 continues until the TCP sender receives an
   ACK acking a sequence number greater than SS_PTR, or it receives
   a SACK block covering the sequence number greater than SS_PTR.

   If the sender receives a SACK block containing SS_PTR, i.e., if
   there is a packet loss in the stalled window, it SHOULD go to
   step-8.

   If the sender receives an ACK that acknowledges SS_PTR, i.e., if
   no packets were lost from the stalled window, it SHOULD go to
   step-10.

NOTE: In our previous experiments we had set the congestion window
   to one MSS after a spurious timeout, however this algorithm prerforms
   better if there is moderate load on the routers and the number of
   competing connections do not vary a lot duing the stalling period. In
   case of heavy load, setting the congestion window to INIT_WINDOW
   still performs better. We believe that using the present congestion
   response make a fair compromise for different scenarios.

[3.3](#) Timeout-Recovery: recovering lost packets after timeout

8. The TCP sender traverses the retransmission queue and marks
   all the packets without any SACK tag as lost. The TCP sender
   also updates its packets-in-flight (pipe) based on the SACK tags
   and the lost segment information (the packets-in-flight (pipe)
   should be ZERO after the update).

[draft-swami-tsvwg-tcp-dclor-01.txt](#)   Apr 2003

Please note that unlike Fast-Retransmit and Fast-recovery, DCLOR

uses only one SACK block containing SS_PTR to mark packets as
lost. This is because we do not expect packet reordering to
exist over the period of RTO.

   9. The TCP sender should update its SS_THRESH, as:

          SS_THRESH= stale_outstanding >> 1   (step-2)

   10. The TCP sender SHOULD set its congestion window to cwnd+1.
       If packets were lost into the network (i.e., if a SACK for
       SS_PTR was received), the TCP sender should start by sending
       packets with lowest sequence number; else it should continue
       with new data. (Note: for each new SACK block received, the
       sender should send a segment--lost or new--and therefore the
       problem of duplicate ACKs is not of concern here.)

       The sender should follow the normal window growth strategy based
       on the value of SS_THRESH after this step.

   Please note that with a pure ACK acknowledging SS_PTR, the TCP sender
   does not update the SS_THRESH value (it directly enters step-10 from
   step-7). This prevents a TCP sender from setting its SS_THRESH to a
   very small values if the spurious timeout occurs at the start of the
   connection.

4. Data Delivery To Upper Layers

   If a TCP sender loses its entire congestion window worth of data,
   sending new data after timeout prevents a TCP receiver from
   forwarding the new data to the upper layers immediately. However,
   once the SACK for this new data is received, the TCP sender will send
   the first lost segment. This essentially means that data delivery to
   the upper layers could be delayed by at most one RTT when all the
   packets are lost in the network.

   This, however, does not affect the throughput of the connection in
   any way. If a timeout has occurred, then the data delivery to the
   upper layers has already been excessively delayed. Delaying it by
   another round trip is not a serious problem. Please note that
   reliability and timeliness are two conflicting issues and one cannot
   gain on one without sacrificing something else on the other.

5. Security Considerations

   The TCP SACK information is meant to be advisory, and a TCP receiver
   is allowed--though strongly discouraged--to discard data blocks the
   receiver has already SACKed [RFC2018]. Please note however that even

if the TCP sender discards the data block it received, it MUST still send the SACK block for at least the recent most data received. Therefore in spite of SACK reneging, DCLOR will work without any deadlocks.

A SACK implementation is also allowed not to send a SACK block even though the TCP sender and receiver might have agreed to SACK- Permitted option at the start of the connection. In these cases, however, if the receiver sends one SACK block, it must send SACK blocks for the rest of the connection. Because of the above mentioned leniency in implementation, its possible that a TCP receiver may agree on SACK-Permitted option, and yet not send any SACK blocks. To make DCLOR robust under these circumstances, DCLOR SHOULD NOT be invoked unless the sender has seen at least one SACK block before timeout. We, however, believe that once the SACK-Permitted option is accepted, the TCP sender MUST send a SACK block--even though that block might finally be discarded. Otherwise, the SACK-Permitted option is completely redundant and serves little purpose. To the best of our knowledge, almost all SACK implementations send a SACK block if they have accepted the SACK-Permitted option.

6. References

   [RFC2581] M. Allman, V. Paxson, W. Stevens. "TCP Congestion
             Control," Apr, 1999.

   [RFC2914] S. Floyd, "Congestion Control Principles," Sep 2002.

   [RFC2861] M. Handley, J. Padhye, S. Floyd. "TCP Congestion
             Window Validation," Jun 2000.

   [BAFW03] E. Blanton, M. Allman, K. Fall, L. Wang, "Conservative
             SACK-based Loss Recovery Algorithm for TCP," draft-
             allman-tcp-sack-13.txt. Internet draft; work in progress.
             Oct 2002.

   [RFC2018] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP
             Selective Acknowledgment Options," Oct 1996.

   [RFC2883] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, "An
             Extension to the Selective Acknowledgment (SACK) Option
             for TCP," Jul 2000.

   [LM02]   R. Ludwig, M. Meyer. "The Eiffel Detection Algorithm

for TCP." Internet draft; work in progress, draft-ietf-tsvwg-tcp-eifel-alg-07.txt, Dec 2002.

[LG03]   R. Ludwig, A. Gurtov, "The Eifel Response Algorithm for
           TCP." Internet draft; work in progress, draft-ietf-tsvwg-tcp-eifel-response-03.txt, Mar 2003.

[SK03]   P. Sarolahti, M. Kojo. "F-RTO: A TCP RTO Recovery
           Algorithm for Avoiding Unnecessary Retransmissions."
           Internet draft; work in progress. draft-sarolahti-tsvwg-tcp-frto-03.txt, Jan 2003.

[RFC2988] V. Paxon, M. Allman. "Computing TCP's Retransmission
           Timer," Nov 2000.

[BA02]   E. Blanton, M. Allman, "Using TCP DSACKs and SCTP
           Duplicate TSNs to Detect Spurious Retransmissions,"
           Internet draft; work in progress, draft-blanton-dsack-use-02.txt, Oct 2002.

7. IPR Statement

   The IETF has been notified of intellectual property rights claimed in
   regard to some or all of the specification contained in this
   document. For more information consult the on-line list of claimed
   rights at http://www.ietf.org/ipr.

Author's Address:

   Yogesh Prem Swami   Khiem Le
   Nokia Research Center   Nokia Research Center
   6000 Connection Drive   6000 Connection Drive
   Irving TX-75063   Irving TX-75063
   USA   USA

   Phone: +1 972-374-0669   Phone: +1 972-894-4882
   Email: yogesh.swami@nokia.com Email: khiem.le@nokia.com