

Network Working Group  
Internet-Draft  
Expires: March 31, 2006

Y. Swami  
K. Le  
Nokia Research Center, Dallas  
September 27, 2005

Decorrelated Loss Recovery (DCLOR) Using SACK Option for Spurious  
Timeouts  
draft-swami-tsvwg-tcp-dclor-06

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

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

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

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

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

This Internet-Draft will expire on March 31, 2006.

Copyright Notice

Copyright (C) The Internet Society (2005).

Abstract

A spurious timeout in TCP forces the sender to unnecessarily retransmit one complete congestion window of data into the network. In addition, the congestion state of the network could change substantially after a spurious timeout. In this draft we propose a conservative congestion response algorithm after spurious timeout that takes network state into account.

Internet-Draft

TCP DCLOR

September 2005

## 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. TCP therefore assumes that a timeout is a sufficient indication to a) recover all the packets in flight, and b) to initiate a congestion response (slow start in this case) suited for heavy congestion scenarios.

Although the assumption that a timeout can occur only if there is severe congestion is valid for traditional wireline 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. In cellular networks, for example, the link layer can experience a relatively long disruption due to errors, and the link layer protocol can keep all packets 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 after a spurious. The algorithm described here follows the congestion control principle of [1] [3] and [5], but unlike the present go-back-N loss recovery algorithm after timeout, DCLOR only sends those segments that were actually lost in the network.

---

Internet-Draft

TCP DCLOR

September 2005

## 2. Terminology

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," "OPTIONAL," and "silently ignore" in this document are to be interpreted as described in [RFC 2119](#).

---

  
Internet-Draft

TCP DCLOR

September 2005

### 3. Problem Description

Let us assume that a TCP sender has sent  $N$  packets,  $p(1) \dots p(N)$ , into the network and it's waiting for the ACK of  $p(1)$ . Due to bad network conditions or some other problem, these packets are excessively delayed at some intermediary node RTR-1. This excessive delay forces the TCP sender to timeout and enter slow start.

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 sender retransmits  $P1(1)$  and waits for the ACK  $a(1)$  ( where  $Px(k)$  represents the  $x$ th retransmission of packet with sequence number  $k$ ).

After some period of time when the network conditions at RTR-1 improve, the queued in packets are finally dispatched to their intended recipient. In response, 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. Here we only analyze the congestion control behavior after a spurious timeout. Our scheme can be used in conjunction with the detection schemes in [6] and [9].

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 timeout (real or spurious), 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, the conservation of packet principle has been compromised, and of the  $3N/2$  packets injected in the network,  $N/2$

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.

Now let's assume there are  $M$  competing TCP connections that share the same bottleneck router(s) with  $C(0)$  (each connection is numbered  $C(0) \dots C(M-1)$ ). During the period of time while  $C(0)$  is stalled, the TCP sender does not use its network resources--the buffer space--on the bottleneck router(s). The competing connections,  $C(1) \dots C(M)$ , 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 following a spurious timeout can be as high as:  $N/2 + M*k$ .

The Eifel response algorithm [7] solves the problem of  $N/2$  packet

loss, by restoring the congestion window to an old value immediately before the spurious timeout. Based on the above equation, however, we note that the congestion state of the network not only depends upon the old window size, but also upon the duration of spurious timeout. In our response algorithm, we therefore take the time duration of spurious timeout into account by reducing the data rate by half every RT0. 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 addition to the above congestion and packet loss issues, the current response after spurious timeouts is inefficient, in the sense that it unnecessarily retransmits data that is not lost, but simply stalled. Such unnecessary retransmission is an issue when bandwidth resources are at a premium, like over a cellular link, where spectrum is scarce and expensive.

#### [4.](#) DCLOR Response Algorithm

A TCP sender should follow [\[6\]](#) or [\[9\]](#) (or any other algorithm) to detect a spurious timeout. If the spurious timeout is confirmed and the TCP SACK option [\[4\]](#) is enabled, only then it SHOULD follow the DCLOR algorithm.

The basic idea of this algorithm is that the ACKs received for the stalled packets don't provide sufficient information about the end-to-end congestion state of the network. Therefore, the sender reduces the congestion window by 1/2 every RT0, and waits for the ACK or SACK of a new data packet before increasing it's congestion window. Additionally, while the sender is waiting for the ACK/SACK of new data, it's allowed to send cwnd (the updated cwnd) worth of new data into the network.

1. The TCP sender MUST record the time when the first timeout took place, and when the first ACK after the timeout was received. Based on these times (or through some other means) it should compute the number of unbacked-off timeouts that must have taken place during this time period. Let's call this number  $N\text{-RTO}$ . The sender should also keep the highest sequence number of data packet that was sent in a variable called `SS_PTR`. The sender should also keep a counter called `dclor_cntr`, which allows the sender to send new data while it's waiting for the ACK or SACK of `SS_PTR`. Additionally, the sender MUST NOT update the `SS_THRESH` value due to spurious timeouts (i.e., the spurious timeout algorithm should leave `SS_THRESH` values unaltered).
2. Once the Spurious Timeout is confirmed, the TCP sender should set  $\text{cwnd} = \max(2, \text{pipe-size}/2^{N\text{-RTO}})$ . (where `pipe-size` is the packets in flight at the time when spurious timeout was confirmed.) Additionally, it should set `dclor_cntr = 0`.
3. For each ACK or SACK  $< \text{SS\_PTR}$  (i.e., a SACK block whose left edge is  $< \text{SS\_PTR}$ ), the sender SHOULD send one \*new\* data packet if it is present and if `dclor_cntr`  $< \text{cwnd}$  and ( $\text{rwnd} < \text{SND.NXT} - \text{SND.UNA}$ ). If ( $\text{rwnd} \geq \text{SND.NXT} - \text{SND.UNA}$ ) or if there is no new data to send, then the sender MUST retransmit no more than one packet per RT0 from the tail of the retransmission queue regardless of the value of `dclor_cntr`. Moreover, for each \*new\* packet sent, `dclor_cntr` should be incremented by one. For ACK/SACK  $< \text{SS\_PTR}$ , the sender MUST not initiate any loss recovery algorithm nor should it update `cwnd` value. Additionally, the `SS_THRESH` should be left unchanged for all these ACKs.

4. If the sender receives a pure ACK  $> \text{SS\_PTR}$ , it should update  $\text{cwnd} = \text{cwnd} + 1$ , and follow normal TCP behavior. (Note that this means that none of the stalled packets were lost so we don't need to change `SS_THRESH` value).
5. If the sender receives a SACK block whose left edge is greater than `SS_PTR`, then it should traverse the retransmission queue from `SND.UNA` to the left edge of SACK block, and mark all

unsacked packets as lost. Additionally, it should set `cwnd = cwnd + 1` and reset `SS_THRESH` to 1/2 the pipe-size. Beyond this point, the sender MUST recover lost packets based on [2].



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.

## 6. SACK renegeing

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 [4]. Please note however that even if the TCP receiver 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 renegeing, 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 receiver 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.

Internet-Draft

TCP DCLOR

September 2005

## [7.](#) Security Consideration

DCLOR does not open TCP to new attacks.

---

Internet-Draft

TCP DCLOR

September 2005

## [8.](#) Acknowledgments

We would like to thank Shashikant Maheshwari, Pasi Sarolahti, and Mika Liljeberg for their comments and suggestions on a previous version of this draft. Special thanks to Jani Hirsimaki for thoroughly reviewing the document and providing feedback on the algorithm.

## [9.](#) References

- [1] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", [RFC 2581](#), April 1999.
- [2] Blanton, E., Allman, M., Fall, K., and L. Wang, "Conservative SACK-based Loss Recovery Algorithm for TCP", [RFC 3517](#), April 2003.
- [3] Floyd, S., "Congestion Control Principles", [RFC 2914](#), September 2002.
- [4] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "TCP Selective Acknowledgement Options", [RFC 2018](#), July 2000.
- [5] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", [RFC 2861](#), June 2000.
- [6] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm", [RFC 3522](#), April 2003.
- [7] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP.", Internet draft; work in progress, [draft-ietf-tsvwg-tcp-eifel-response-05.txt](#), March 2004.

- [8] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", [RFC 2988](#), November 2000.
- [9] Sarolahti, P. and M. Kojo, "F-RTT: A TCP RTT Recovery Algorithm for Avoiding Unnecessary Retransmissions.", Internet draft; work in progress, July 2004.

#### Authors' Addresses

Yogesh Prem Swami  
Nokia Research Center, Dallas  
6000 Connection Drive  
Irving, TX 75039  
USA

Phone: +1 972 374 0669  
Email: [yogesh.swami@nokia.com](mailto:yogesh.swami@nokia.com)

Khiem Le  
Nokia Research Center, Dallas  
6000 Connection Drive  
Irving, TX 75039  
USA

Phone: +1 972 894 4882  
Email: [khiem.le@nokia.com](mailto:khiem.le@nokia.com)

#### Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

#### Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

#### Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

#### Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.