Internet Engineering Task Force
INTERNET DRAFT
File: draft-swami-tsvwg-tcp-dclor-00.txt

Yogesh Swami
Khiem Le
Nokia Research Center
Dallas

November 2002
Expires: Apr 2003

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 instead of relieving it.

In this draft we propose changes to the TCP sender (no change is needed for TCP receiver) that can be used to solve the problems of

both redundant-retransmission and network congestion after a spurious timeout. These changes preserve the sanctity of congestion control principles and are conservative in nature. The proposed algorithm--called DCLOR--separates congestion control from loss recovery, and uses the TCP SACK option to achieve this. DCLOR can be used as a congestion control algorithm (after a spurious timeout) only, and it can work with other spurious timeout detection mechanisms such as the Eifel detection scheme.

Table of Contents

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.

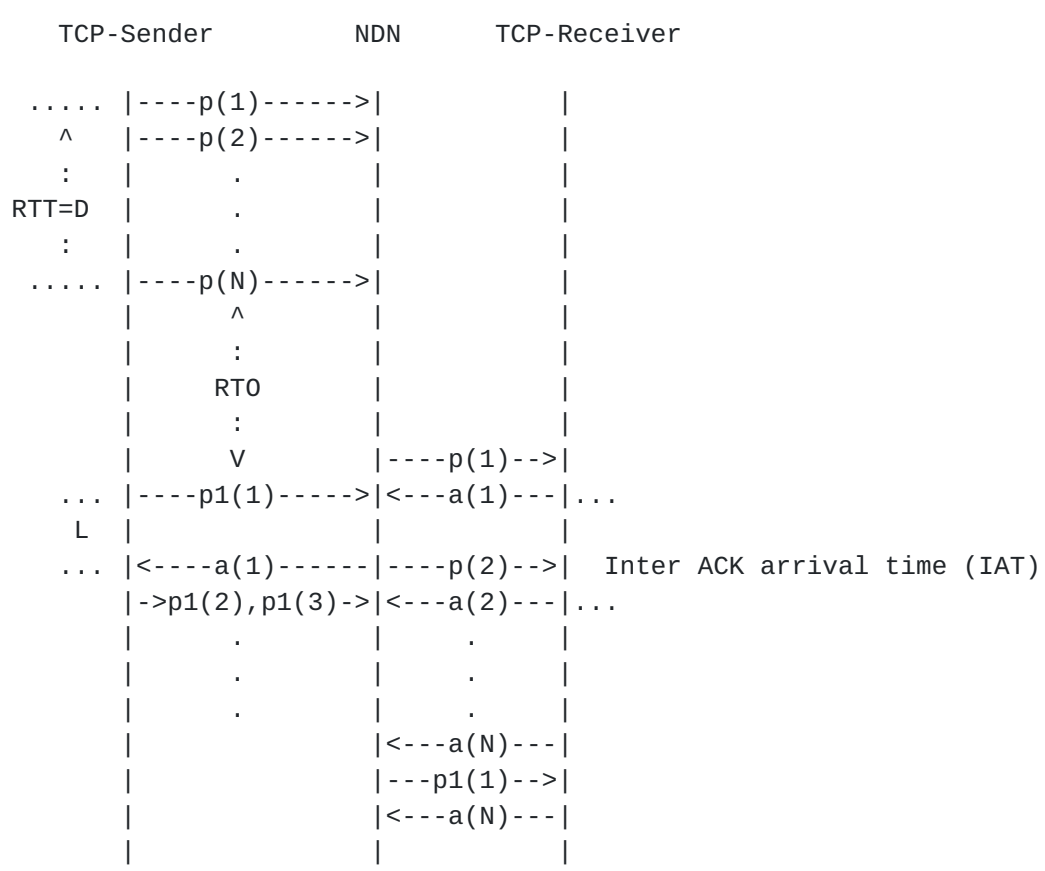
In this document we present an alternative approach to loss recovery and congestion control that "De-Correlates" Loss Recovery from congestion congestion and allows independent choice on using a particular TCP sequence number without compromising on the congestion control principles of [1][2][3][4]. In doing so, the algorithm mitigates the ill effects of spurious retransmission timeouts. Spurious timeouts are not only detrimental to the throughput of the defaulting connection, but they also add to the overall network congestion--paradoxically enough, due to the subsequent slow-start.

Although several drafts [7][8] have been presented on this topic in the IETF, 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-4.

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



As far as the TCP 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 the network conditions at NDN become amicable again, and the queued in packets at NDN are finally

dispatched to their intended recipient, to which the the TCP receiver generates the ACK a(1). When the TCP sender receives a(1), it's fooled into believing that this ACK 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 subsections 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 Can Slow Start add to Congestion after Spurious Timeout

Paradoxically, slow start--the epitome of congestion control--can itself add to network congestion after a spurious timeout. Going back to the previous example, when the TCP sender receives a(1), it falsely assumes that a(1) was triggered by p1(1). But in reality, a(1) was triggered by p(1), not p1(1). This ambiguity exits because of the cumulative acknowledgment property of TCP. In this case, if the TCP sender were to take RTT samples (which it does not take because of ACK ambiguity), it would register a value of L (Figure-1) instead of D--the RTT. In addition to this, when the TCP sender sends a1(2) and a1(3), it will receive the ACK for a1(2) and a1(3) right after the inter ACK arrival time IAT. Then-after, the TCP sender will keep increasing the data rate every IAT as new ACKs arrive. Mathematically, one can show that after a spurious timeout, the rate at which data is injected in to the network is given by 1. But if we want to preserve the network probing capability [2] of a TCP sender, then the rate should be given by (2).

$$r(t) = 2*ceil(t-t0/IAT) \quad \ldots (1)$$

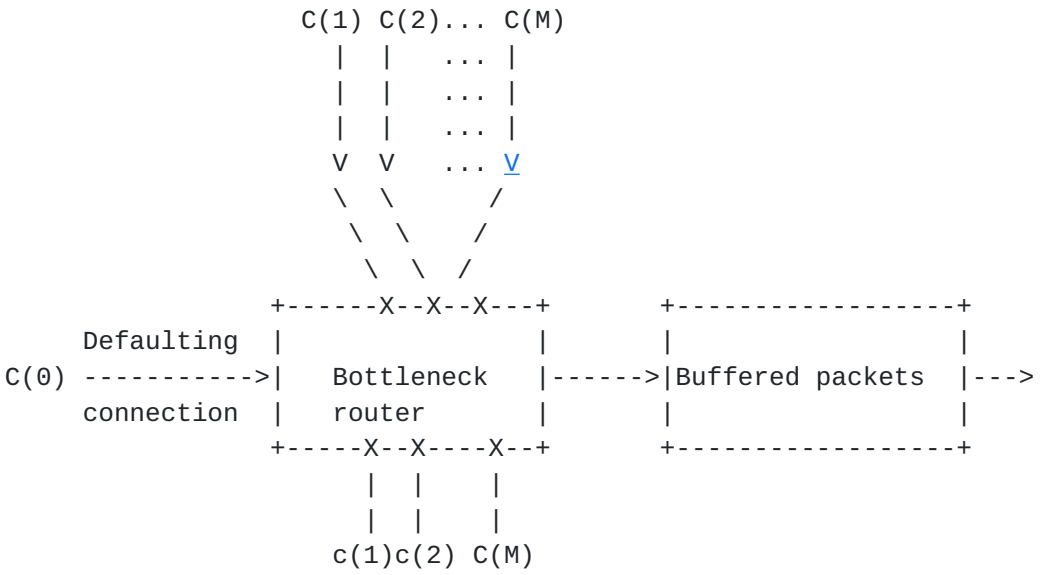
$$r(t) = 2^{ceil(t-t0/D)} \quad \ldots (2)$$

Where t>t0, and t0 is the time when first ACK after spurious timeout is received, and IAT and D are related by (3)

IAT <= D/N ... (3)

(N is the flight size at the time of spurious timeout).

Figure-2



We now compute the worst-case packet loss due to this data burst. After the timeout, the TCP sender will set its SS_THRESH to N/2 (packets-in-flight/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 also 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. The number of data segments that left the network during this time is only N. Therefore, there is a high probability that N/2 packets out of 3N/2 packets will be lost. 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 as the defaulting connection C(0) does (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). If, for simplicity reasons, we assume that each of these TCP connections has the same round trip time of RTT, and if the idle time for C(0) is k*RTT, (where k > RTO/RTT) then 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:

$$N/2 + M*k \quad \ldots (4)$$

where the first term is the packet loss due to slow start, while the second term is the loss due to window growth of completing connections.

Please note that even if a TCP sender restores its congestion window [7](or halves [8] it) to avoid slow start and redundant retransmission, it still cannot avoid the loss of M*k segments (M*k-N/2 segments in case the window was halved) that were added in the network because of sender's inactivity. Since a TCP sender does not know the number of connections it's competing with, or the time duration for which packets could be stalled in the network, the number of segments lost due to spurious timeout could be large.

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

3. Sources of spurious timeout

Since the response algorithm after a timeout depends on the event that triggered it, it is important to know the factors that can cause such timeouts (in addition to the timeouts caused by congestion.) The following list enumerates few broad categories of events that can cause spurious timeouts in TCP.

3.1 Spurious timeout due to Excessive delay at a router

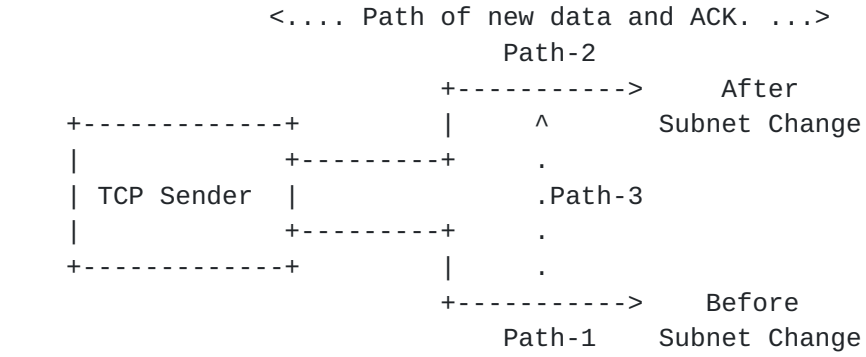
Many link layer technologies have their own loss recovery schemes that use link layer feedback mechanisms to recover bit losses. These loss recovery schemes are often unaware of the transport layer loss

recovery schemes. Therefore, if there is burst of error at the link layer, the TCP sender may timeout because of
the loss recovery scheme used by the link layers.

3.2 Spurious timeout due to change of subnets

A TCP receiver or sender may change subnets due to sender or receiver mobility (Figure-3). While the change of subnets is taking place, a router may keep the data packets buffered until a new route is established (i.e., until Path-3 and Path-2 are established in Figure-3). Since establishing a new route can take a long time, a TCP sender may timeout.

Figure-3



Once the new path is established, the router may forward all data packets to the TCP receiver using Path-3. However, the ACKs triggered by these packets will reach the TCP sender using Path-2. In addition, the TCP sender will send new data in response to these ACKs on Path-2.

When there is a change of subnets, the entire end-to-end path between the sender and receiver might change completely (for example, Path-1 and Path-2 may not have any common routers between then). A complete change in the end-to-end path may take place, for example, if a TCP receiver uses Mobile-IPv6 with route optimization to move from one subnet to another. Since the TCP connection does not have any information about the BDP on new path (Path-2), the TCP connection should start with a window size same as the window size of a new connection, i.e., with a size of two.

Please note that of all possible reasons for spurious timeout, a change in subnets has maximum impact on network congestion. Since the BDP on a new path could be tens of orders of magnitude higher or lower than the BDP on an old path, an inappropriate congestion

control scheme could severely add to network congestion on the new path (Path-2). For example, let us assume there are 100 TCP connections sharing the same bottleneck router. Let's also assume that the average number of subnet change per second per TCP receiver is 1/50, and the average congestion window for each TCP connection at the time of spurious timeout is 5 segments. Then in the worst case, the network may drop as many as (1/50)*(5/2)*100 = 20 segments every second if the congestion window is restored as done in [7].

3.3 Spurious Timeout due to Protocol Inefficiencies

A TCP sender may timeout because of inherent protocol inefficiencies in TCP. For example, a spurious timeout may occur if multiple packets are lost and fast-retransmit/fast-recovery algorithms could not recover the lost packets within an RTO. A spurious timeout may also occur if there are so few packets injected into the network that in case of a packet loss the receiver cannot generate 3 duplicate ACKs.

4. De-correlated Loss Recovery (DCLOR)

De-correlated loss recovery tries to remedy the problems described in Section-2 by separating congestion control from loss recovery mechanism. In doing so, we make the decision of "which packets to send" independent from "how much" data to send. In addition to this, the rate at which data is injected into the network preserves the conservation control principles as described in [1][2][3].

The basic idea behind DCLOR is to send a new data segment from outside the sender's retransmission queue--the queue of unacknowledged data segments--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 in case of slow-start, but only tries to recover those segments that were lost in the network. This not only allows the TCP sender to avoid the go-back-N problem, but also prevents superfluous congestion window growth as seen with slow-start (section-2.2).

Before describing the algorithm itself, we fist enumerate the underlying design philosophy behind DCLOR as it can give more insight into the workings of the algorithm:

1. A spurious timeout should not degrade the performance of

other TCP connections sharing the same network path. As pointed out in Section-2, the slow-start subsequent to spurious timeout not only affects the performance of the defaulting connections, but it can also confer data-loss on other competing connections.

2. Congestion control decisions should be made independent of the loss recovery decision. In other words, the congestion control algorithm should only determines the amount of data to be injected into the network, while the loss recovery algorithm should determine the sequence number of data to be recovered.

3. Since the state of the network can change dramatically after a long interruption--long with respect to RTT--(Section-3), the response algorithm after a timeout should be conservative in the amount of data injected into the network after a timeout. This avoids packet loss (and congestion) due to inappropriately inflated congestion window after spurious timeouts (Section-2.2)

4. The TCP sender's estimate of network capacity--the SS_THRESH--should be updated if and only if packets are lost in the network. A spurious timeout without packet loss should not affect the SS_THRESH since SS_THRESH is a measure of the network's buffering capacity and there is no need to updated it unless packet loss is detected.

5. If there is packet loss in addition to packet stalling, all the lost packets within that window should be recovered without any need for future timeouts. In other words, the algorithm should not require the TCP sender to timeout again due to protocol limitations.

We now describe the algorithm in more detail, keeping the above mentioned points in mind. Please see Figure-4 for certain parameters.

4.1 Probe phase after a timeout

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

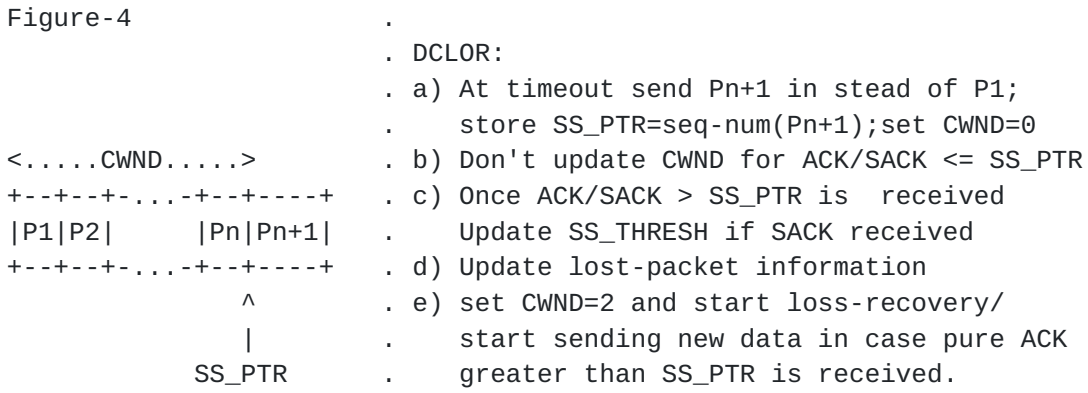
1. If the timeout occurs before the 3 way handshake is complete, the TCP sender's behavior is unchanged,

2. After a timeout, the TCP sender MUST set its congestion window to ZERO (CWND = 0), and keep the number of outstanding packets (N)into a state variable for future use (step-9). The value of SS_THRESH MUST be left UNCHANGED at this point.

3. The TCP sender SHOULD also reset all the SACK tag bits in its retransmission queue (essentially renege all the SACK information in accordance with the recommendation in [5]. Please also see Section-6 on SACK reneging for more information.)

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

Figure-4



TCP Sender's Retransmission Queue

The TCP sender also stores the sequence number of this new data packet 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).

The idea behind sending one packet on a timeout is to probe the network and determine if the network is congested. However, from the network point of view it does not matter what the sequence number of data sent was. What matters is the "amount" of data sent into the network. Therefore, sending a new packet from outside the retransmission queue has the same affect as sending an old packet. Therefore this behavior is in accordance with the TCP congestion

control and avoidance mechanisms described in [1].

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

4.2 Congestion Control After the probe phase

6. After the timeout, for each ACK received with the ACK-sequence number less than SS_PTR, the packet ACKed is removed from the retransmission queue. If the ACK contains a new SACK block, then the SACK tag is set in the corresponding data packet. However, no data is sent for these ACKs.

7. The TCP sender MUST NOT update its congestion window from a value of ZERO until an ACK-sequence number greater than SS_PTR, or a SACK block containing SS_PTR is received. In other words, until the ACK-sequence for Pn+1 is received or a SACK block containing information about Pn+1 is received, NO new data segments (i.e., segments Pn+2 ...,) are added into the network. This allows the TCP sender to discard all the "stale ACKs"--the ACKs that are generated for stalled packets--and prevents superfluous growth of congestion window.

   The TCP sender MUST NOT take RTT samples for stale-ACKs. The SACK information present in stale-ACKs SHOULD be however used to put SACK tags on the retransmission queue. A TCP sender MAY reset its retransmission timer with every stale ACK received.

   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.

The rationale for not sending any data segment for stale ACKs is to minimize congestion after a timeout. Since the duration of packet stalling could be large, the congestion state of the network could have change dramatically. Therefore, not using the stale ACKs to send data allows the TCP sender to recompute its congestion state from scratch and also minimizes packet loss.

NOTE-1: In our experiments, we have tried sending new data for stale ACKs in the spirit of reverting the congestion window as described in [7]. We have also tried setting the congestion window to

half the packets-in-flight (pipe) as described in [8]. But we have found that not sending any new data, as described above, is the best scheme whenever there are multiple competing connections in the system. The results of these experiments, along with the experimental setup, are listed in Appendix-1.

If a TCP sender has not received any ACK within an RTT, sending more than a few data segments (>4) will add to network congestion in one form or the other (in varying degrees) and the overall performance of the system as a whole degrades--on an average. In addition, the DCLOR scheme works well in case of subnet change where there could be orders of magnitude difference in BDP. DCLOR also has minimal effect on other competing connections.

Based on our experiments, we do not recommend sending new data for stale ACKs, but it's possible to deploy a variety of congestion control schemes with the DCLOR framework. For example, a TCP sender on the reception of an old ACK MAY revert (or halve) its congestion window to the old value of packets-in-flight (pipe) and MAY send new data depending upon the congestion window. But the TCP sender SHOULD initiate its loss recovery ONLY after SS_PTR has been ACKed or SACKed. In other words, if the TCP sender receives duplicate ACKs with ACK-sequence number less than SS_PTR, it SHOULD NOT use these duplicate ACKs to initiate Fast-Retransmit and Fast-Recovery. The loss recovery SHOULD be initiated only after ACK/SACK for SS_PTR is received. After that the sender should follow the same steps as described in step-8, step-9, and step-10 (with a different congestion window of course).

NOTE-2: For the congestion response of DCLOR, we have also considered adaptive update of congestion window after each timeout. In this scheme a TCP sender reduces its congestion window by a factor of 2 for every RTO period of inactivity. The SS_THRESH is not updated until a loss is detected. At the time of this writing we are still experimenting with the usefulness of such a scheme. However, such a scheme is not well suited for spurious timeouts caused by change of subnets as described in Section-3.2.

4.3 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).

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 updates its SS_THRESH, as following:

   SS_THRESH=packets-in-flight (pipe) at the time of timeout / 2

   Please note that the TCP sender stored the value of packets-in-flight at the time of timeout in step-2.

10. The TCP sender sets its congestion window to 2. If packets were lost into the network (i.e., if a SACK for SS_PTR was received), the TCP sender should start by sending the least sequence number packets, else it should continue by sending new data.

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 value if the spurious timeout occurs at the start of the connection.

The rationale behind not updating the SS_THRESH if no loss is detected is that SS_THRESH represents the sender's estimate of network capacity--the BDP. Unless a loss is detected, there is no reason to update the value of BDP.

5. Data Delivery To Upper Layers

If a TCP sender loses its entire congestion window worth of data due to congestion, 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 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 Reneging

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[5]. 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.

7. DCLOR Examples

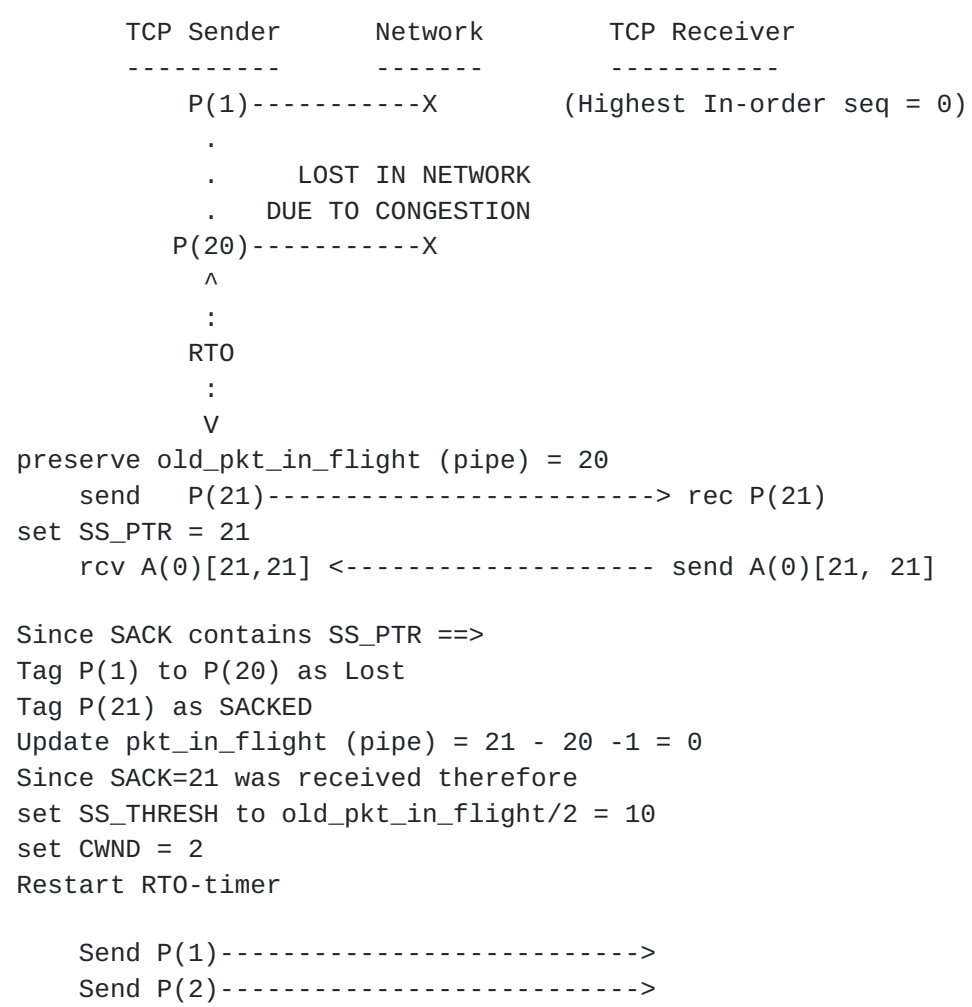
We now demonstrate the working of DCLOR with a few concrete examples. We first take the case in which the timeout is caused due to congestion. We then consider the case, in which all the packets are stalled in the network but there is no packet loss. Finally, we take the example in which packets are both lost and stalled at the same time.

In all these examples, the TCP sender has 20 MSS packets in flight (pipe) at the time when timeout occurs and each packet is denoted by P(i) (please note that we are using packet numbers instead of TCP sequence numbers to make the presentation simple. The example can be modified to take TCP sequence numbers into account too.) An ACK is denoted by A(i)[x,y][z,w], where i is the highest packet number acknowledged, and the ordered pair [x,y] and [z,w] represent the sack blocks present in that ACK (please note that $i < x,y,z,w$ and its possible that x=y if just one packet is being sacked.)

7.1 Timeout due to congestion.

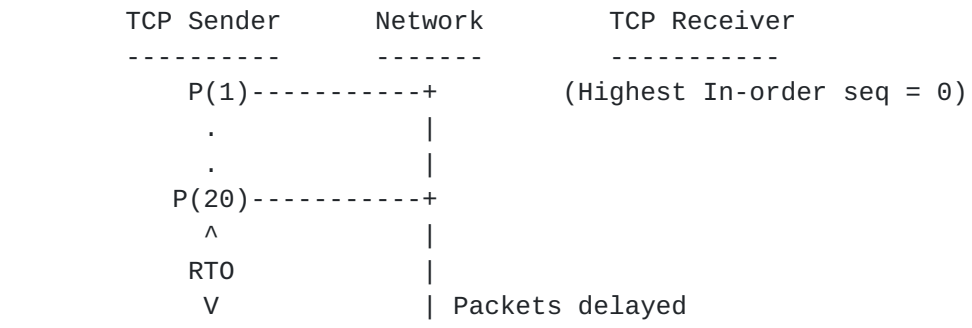
In this case, we assume that all packets P(1) to P(20) are lost into

the network. The following time-line shows the sequence of events:



7.2 Timeout due to pure packet stalling.

In this case, we assume that all the packets P(1) to P(20) are stalled in the network. The following time-line shows the sequence of events:



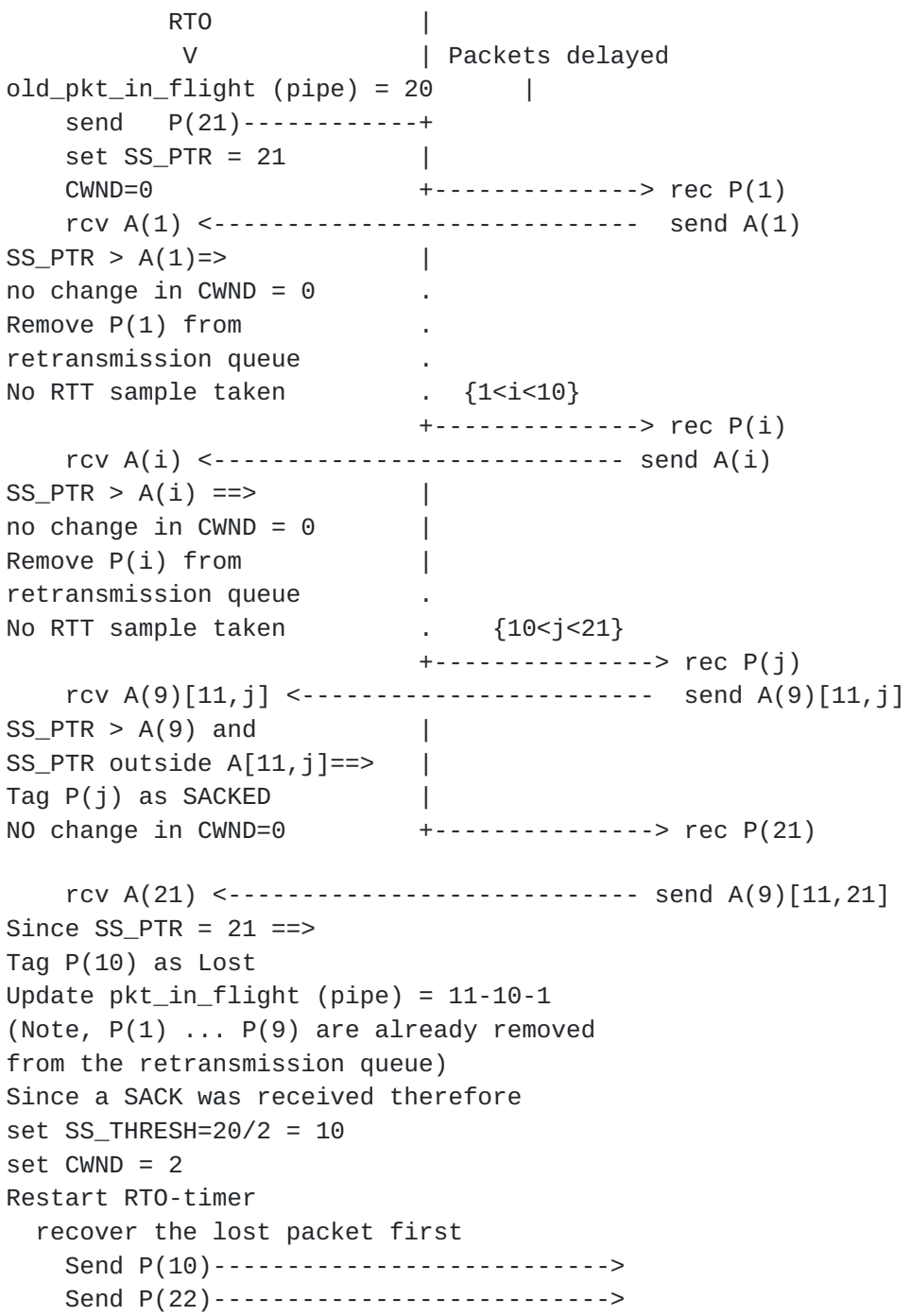
```
old_pkt_in_flight (pipe) = 20    |
    send   P(21)----------+
    set SS_PTR = 21       |
    CWND=0                +--------------> rec P(1)
    rcv A(1) <---------------------------- send A(1)
SS_PTR > A(1)=>           |
no change in CWND = 0     .
Remove P(1) from          .
retransmission queue      .
No timer sample taken     .
                          +--------------> rec P(i) {1<i<21}
    rcv A(i) <---------------------------- send A(i)
SS_PTR > A(i) ==>         |
no change in CWND = 0     |
Remove P(i) from          |
retransmission queue      .
No timer sample taken     .
                          +---------------> rec P(21)
    rcv A(21) <---------------------------- send A(21)
SS_PTR = A(21) ==>
pure ACK received
Remove P(21) from retransmission queue
Update pkt_in_flight (pipe) = 0
Since Pure ACK was received therefore
SS_THRESH remains unchanged.
set CWND = 2
Restart RTO-timer
    Send P(22)--------------------------->
    Send P(23)--------------------------->
```

7.3 Comprehensive Scenario: Timeout due to stalling and loss.

Building upon the previous two example, we now assume that in addition to data stalling, P(10) was lost into the network.

The following time-line shows the sequence of events:

```
        TCP Sender      Network        TCP Receiver
        ----------      -------        ------------
            P(1)-------------+      (Highest In-order seq = 0)
             .               |
           P(10)----X Lost   |
             .               |
           P(20)-------------+
             ^               |
```



8. Security Considerations

No new security threats are introduced by the use of DCLOR.

9. References

[1] M. Allman, V. Paxson, W. Stevens. "TCP Congestion Control. RFC-2581."

[2] S. Floyd. "Congestion Control Principles." RFC-2914.

[3] M. Handley, J. Padhye, S. Floyd. "TCP Congestion Window Validation." RFC-2861.

[4] Ethan Blanton, Mark Allman, Kevin Fall. "Conservative SACK-based Loss Recovery Algorithm for TCP." draft-allman-tcp-sack-10.txt. Work in Progress.

[5] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow. "TCP Selective Acknowledgment Options." RFC-2018.

[6] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky. "An Extension to the Selective Acknowledgment (SACK) Option for TCP" RCF-2883.

[7] Reiner Ludwig, Michael Meyer. "The Eiffel Detection Algorithm for TCP." Internet Draft-work in progress. draft-ietf-tsvwg-tcp-eifel-alg-05.txt.

[8] 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-01.txt

[9] V. Paxon, M. Allman. "Computing TCP's Retransmission Timer." RFC-2998

10. IPR Statement

Author's Address:

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

USA

Phone: +1 972-374-0669
Email: yogesh.swami@nokia.com

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

Phone: +1 972-894-4882
Email: khiem.le@nokia.com

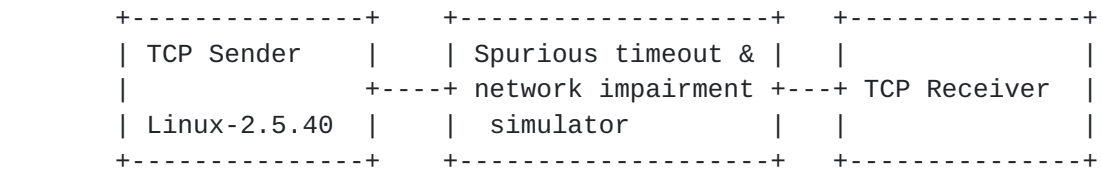
Appendix - 1

In this section we provide information on our test setup and provide test results for comparison of DCLOR with Standard TCP, Eifel, and FRTO schemes.

App-1: Testing Methodology

Please see Figure-5 for the test setup. The TCP sender and TCP receiver are connected through an intermediary "spurious timeout and network impairment simulator (STIS)," that simulates packet delay and certain network impairments. The STIS is capable of reading all TCP packets received on any of its interfaces. It is capable of processing packets coming from multiple TCP connections. Each TCP packet is then subjected to a series of network impairments that are expected to be present in a typical IP network. The specifics of these impairments along with the model used to simulate them is enumerated below:

Figure-5



1. Bottleneck Router

   The STIS is capable of simulating a bottleneck router on the end-to-end path of a TCP connection. For TCP, a bottleneck router is one that has the minimum ratio for the buffer-space to difference in link speed on the end-to-end path (i.e., if b(i) represents the buffer space of router i, and a(i) and d(i) be the incoming and the outgoing link speed of the router for a particular TCP connection, then the bottle neck router for such a connection is one that has min{ b(i)/[d(i)-a(i)]} for all routers i on the end-to-end path of the TCP connection).

   The bottleneck router is simulated by specifying a buffer space and the outgoing link speed (the arrival rate in our simulation was fixed by the link speed of the interface on which the packet arrived. This speed was a fixed value of 10Mbps). In order to simulate outgoing link, the STIS enqueues each TCP packet and schedules them based on the queuing delay and link-delay (service time) of packets in the queue (please see item-2 for more details). A packet received on any of the interfaces is processed only if the total bytes of enqueued data in the router is less than the specified buffer space. A packet is dropped if the buffer space is full. Please note that the buffer space is allocated for and shared by all possible TCP connections going through STIS.

2. Link Speed Simulation

   Although each TCP connection shares the same buffer space, each TCP connection has its own logical queue for each direction. The departure time of a new packet is computed by addition the link delay (packet-size / outgoing-link-speed) of the new packet to the queuing delay of all packets ahead of it. A TCP sender is scheduled on the appropriate interface once its departure time expires.

3. Fixed Network Delay

   The wired-network delay seen by individual packets is the sum of the queuing delay and link delay of each packet on individual routers. If we assume that the number of routers are infinitely many on the end-to-end path, and if the variance of the delay is bounded (which is the case with finite queue sizes), then by the law of large numbers, the delay seen by each packet will be a constant. Although there are not infinitely many routers in the real networks, to make our model simple we assume that the delay added to each packet on the end-to-end path is a constant. This delay is added in addition to the queuing delay incurred due to

link speed simulation. (Please note that fixed network delay tries to capture the queuing delay on the high speed link, while the delay incurred due to link speed simulation is the delay due to a last hop slow link.)

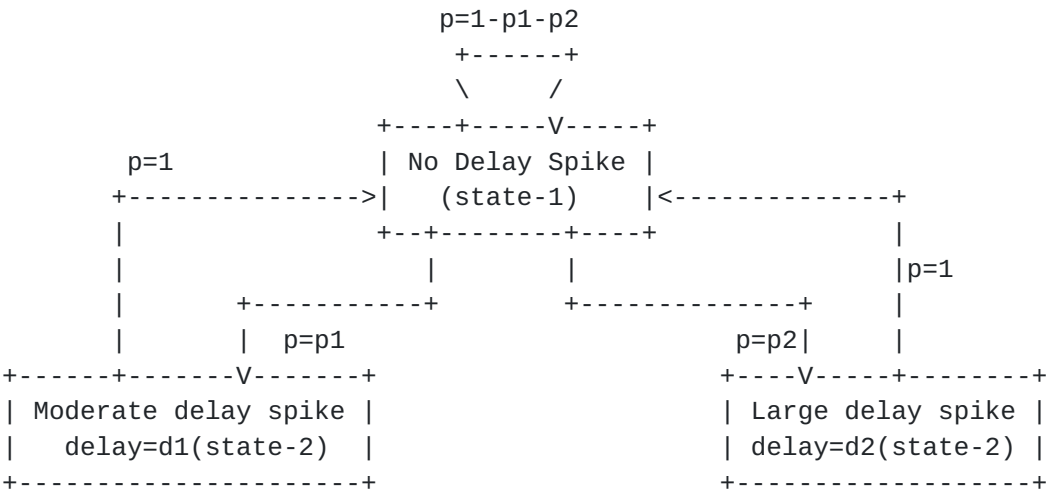
The fixed network delay is simulated by adding a fixed delay to the departure time of each packet.

4. Prolonged packet stalling

In addition to the delay models described above, the STIS also simulates prolonged packet stalling in the network. Although the end result of packet stalling is a delay spike, we do not simply add a long delay burst to achieve packet stalling rather use the Markov model shown in Figure-6. (This model is based on the two different kinds of delay spikes seen in many cellular-networks.)

Every second a random number r (<1) is generated(lets say the random number is generated at time T) for each TCP connection and compared with p1 and p2. If r is less than p1 or P2 a state transition is made into state-2 or state-3. When a TCP connection is in state-2 or state-3, each packet arriving in the time interval T and T+D (where D = d1 or d2 depending upon the state), is treated as if it arrived at time T+D. By altering the arrival time of each packet, this model simulates packet stalling in the network.

Figure-6



Please note that in a real network, when packet stalling occurs--for whatever reason--all packets remain queued in the network until the packets can be forwarded again. Therefore the effect of packet stalling is different from adding a long burst of delay to a few of the packets. Please note that with such a model, the delay incurred by individual packets will not be the same.

Please also note that the state transitions are based on time. Therefore the STIS will keep entering state-2 and state-3 whether or not there are any TCP packets to be forwarded.

5. Packet Reordering

Since the Internet is known to reorder 12% of the packets (this study was done by Vern Paxson of UC Berkeley) on the end-to-end path, we try to simulate this behavior in our simulations. To simulate packet reordering we try to emulate the network behavior that creates packet reordering--that is we try to emulate router failure or load balancing to achieve packet reordering.

A router failure is modeled using a two state Markov model, in which transition from state to another means a route failure. In each of the different states, the TCP connection incurs a different fixed network delay as described above in item-2. This model is based on the assumption that when a route failure takes place, the delay on new path is different from the delay on the old path.

App-2: Test setup, parameters, and Results

Based on the model described above, the STIS uses following parameters for testing.

Table-1: STIS parameters

Parameter Name	Parameter Value
Path MTU	1500 Bytes
Bottleneck Router's Capacity	74K Bytes
Outgoing link data rate	50 K Bits/sec
Fixed Network Delay (one way)	200 ms
Pkt Stalling delay in state-2 (d1)	5 sec
Pkt Stalling delay in state-3 (d2)	8 sec
Pkt Stalling prob to state-2 (p1)	0.05 per-sec

Pkt Stalling prob to state-3 (p2)	0.005 per-sec
Pkt reordering probability	0.12
Pkt reordering delay	20 ms

On the TCP receiver, multiple TCP receiver processes simultaneously run and try to download files of different sizes--each connection competing to download a file of given size. In addition, each process waits for a random time interval and iterates over and over again to download the same file. The following table shows the traffic mix used for our experiments.

Table-2: Traffic Mix

File Size (KB)	# simultaneous connections	# iterations
5	6	2000
10	5	1000
100	5	100
1000	3	10
10000	1	1

Based on the STIS parameters and traffic mix, the cumulative probability distribution of download time, i.e., the plot of number of points with a download time of less than a give time duration in the above experiment, for each file size was generated (because of proper format for representing these plots, we are unable to include these plots in this document. We can provide this plot on request.)

Tables 3,4,and 5 show the mean and variance (in second) for downloading different file sizes using DCLOR, EIFEL, FRTO, and Standard TCP. Please note that not every connection was subjected to a spurious timeout, and therefore the mean value alone is not a good measure of performance. The variance captures the impact of such spurious timeouts (the probability distribution is the best representation of how individual schemes perform for different download time range). As can be seen from the following table, Eifel has a large variance because of restoring the window. Please note that the variance for standard TCP is less than Eifel, or Frto, because the first N packets in case of standard TCP are redundant, and therefore even if they are lost due to error burst, the defaulting connection does not incur a very severe penalty and therefore the variance is small. But for Eifel and FRTO, the N packets after spurious timeout are new and therefore loss of these

packets has a huge impact on the variance.

Table-3: Download Time Statistics for 5K files

(sec)	DCLOR	EIFEL	STD_TCP	FRTO
Mean	2.3869	2.4162	2.3962	2.4049
Variance	3.2473	10.2579	3.2164	4.0466

Table-4: Download Time Statistics for 10K files

(sec)	DCLOR	EIFEL	STD_TCP	FRTO
Mean	3.4547	4.0560	3.7314	3.7317
Variance	4.7452	19.7828	7.6378	8.4576

Table-5: Download Time Statistics for 100K files

(sec)	DCLOR	EIFEL	STD_TCP	FRTO
Mean	24.6297	26.1017	26.7425	25.5325
Variance	66.0804	98.4933	98.9363	78.1667

In addition to the mean and variance of download times, we also calculated the "spectral efficiency" of the system. The spectral efficiency was computed as:

$$SE = \frac{(\text{redundant\_bytes\_received})}{\text{mean(cwnd)*MSS}}$$

The spectral efficiency is a measure of usefulness of a particular

scheme in terms of fraction of extra bytes sent due to spurious timeout for every byte sent into the network. The smaller this number is, the better the scheme is. The numerator is the amount of unnecessary data, while the denominator is the average network capacity experienced by the connection. Since some schemes (Eifel for example) requires more TCP options, the MSS was used instead of MTU in the denominator. Table-5 shows these results for all the four different TCP implementations. Please note that FRTO behaves like standard TCP if there is no new data to send or if the congestion window is as big as flow control window. This is one of the reasons why FRTO performance is worse than Eifel or DCLOR.

Table-6: Spectral Efficiency

File Size	DCLOR (MSS=1460)	EIFEL (1448)	STD_TCP (1460)	FRTO (1460)
5K	0.004042	0.004454	0.092714	0.071336
10k	0.005249	0.012293	0.078977	0.052581
100k	0.017124	0.036748	0.624361	0.079285

The DCLOR algorithm for all these tests cases were implemented in the Linux Kernel-2.5.40 (the source code for DCLOR could be provided on request). The same kernel version was used for all the other TCP enhancements too.