

# Reorder Detecting TCP (RD-TCP) with Explicit Packet Drop Notification (EPDN)

Arjuna Sathiaselan Tomasz Radzik

Department of Computer Science,  
King's College London, Strand, London WC2R2LS, United Kingdom  
Tel: +44 20 7848 2841  
Email: {arjuna,radzik}@dcs.kcl.ac.uk

**Abstract**—Numerous studies have shown that packet reordering is common, especially in networks where there is high degree of parallelism and different link speeds. Reordering of packets decreases the TCP performance of a network, mainly because it leads to overestimation of the congestion of the network. We consider wired networks and analyze the performance of such networks when reordering of packets occurs. We propose a proactive solution that could significantly improve the performance of the network when reordering of packets occurs in the network. We report results of our simulation experiments, which support this claim. Our solution is based on enabling the senders to distinguish between dropped packets and reordered packets.

**Keywords:** TCP, RD-TCP, Throughput, Packet Reorder

## I. INTRODUCTION

Research on the implications of packet reordering on networks indicate that packet reordering is not a pathological network behavior. Reordering of packets occur due to the following reasons:

- Local parallelism: Packet reordering occurs naturally as a result of *local parallelism* [3]: a packet can traverse through multiple paths within a device. Local parallelism is imperative in today's Internet as it reduces equipment and trunk costs.
- Multipath routing: Reordering may result when the packets are routed to a particular destination through multiple paths, each of which may have a different round-trip time (RTT) values [16].
- Frequent route changes: Packet reordering occurs also due to route changes: if the new route offers a lower delay than the old one, then reordering may occur [11].
- Links with high latency: Links such as satellite links have high RTTs, typically on the order of several hundred milliseconds. In order to keep the pipe full, link-layer retransmission protocols send subsequent packets while awaiting an ACK or NAK for a previously sent packet. Here, a link-layer retransmission is reordered by the number of packets that were sent between the original transmission of that packet and the return of the ACK or NAK [17].

A network path that suffers from persistent packet reordering will have severe performance degradation.

## II. TCP PERFORMANCE ON PACKET REORDERING

TCP uses cumulative acknowledgements it receives from the receiver to determine which packets have been successfully received at the receiver and retransmits the packets it believes to have been lost. For example, assume that four segments  $A$ ,  $B$ ,  $C$  and  $D$  are transmitted through the network from a sender to a receiver. When segments  $A$  and  $B$  reach the receiver, it transmits back to the sender an ACK for  $B$  which summarizes that both segments  $A$  and  $B$  have been received. Suppose segments  $C$  and  $D$  have been reordered in the network. When segment  $D$  arrives at the receiver, it sends the ACK for the last in-order segment received which in our case is  $B$ . Only when segment  $C$  arrives, the ACK for the last in-order segment (segment  $D$ ) is transmitted.

Reordering of packets during transmission through the network has several implications on the TCP performance. The following implications are pointed out in [4]:

- 1) When a network path reorders data segments, it may cause the TCP receiver to send three successive DUPACKs, triggering the Fast Retransmit procedure at the TCP sender for data segments that may not necessarily be lost. Unnecessary retransmission of data segments means that some of the bandwidth is wasted.
- 2) The TCP transport protocol assumes congestion in the network when it assumes that a packet is dropped at the gateway. Thus when a TCP sender receives three successive DUPACKs, the TCP assumes that a packet has been lost and that this loss is an indication of network congestion and enters either slow start or the congestion avoidance phase and backs off its retransmission timer (Karn's algorithm) [7]. Moreover if there were multiple fast retransmits within a single window, the cwnd is quickly deflated to a small value. Unnecessary reduction of cwnd caused by packet reordering leads to improper utilization of the link.
- 3) TCP ensures that the receiving application receives data in order. Persistent reordering of data segments is a serious burden on the TCP receiver since the receiver must buffer the out-of-order data until the missing data arrive to fill the gaps. Thus the data being buffered is withheld from the receiving application. This causes unnecessary load to the receiver and reduces the overall efficiency of the system.
- 4) TCP calculates the retransmission time out (RTO) by

sampling and averaging the round trip time (RTT) i.e. the time taken to send a data packet and receive a corresponding acknowledgement for the data packet. When a packet gets reordered in the network, the estimated round trip time is quite large which could falsely inflate the RTO estimation. This has a negative impact on the TCP performance, since if a packet was originally dropped then the TCP has to wait longer to retransmit the dropped packet.

We propose extending the TCP protocol to enable TCP senders to recognize whether a received *dupack* means that a packet has been dropped or reordered. The extended protocol is based on storing at the gateways information about dropped packets. We term this mechanism of informing the sender about dropped packets as Explicit Packet Drop Notification (EPDN). Based on this information, the sender determines whether the packet has been dropped or reordered. The TCP sender then takes the appropriate action based on the information. We call the resulting protocol Reorder Detecting TCP (RD-TCP). The preliminary version of this paper has been presented in [13].

In Section III presents the previous work related to our study. In Sections IV, V and VI we present the details of our proposed solution. In Sections VII and VIII, we describe and discuss the evaluations of our solution via simulations. We conclude the paper with a summary of our work and a short discussion of the further research in Section IX.

### III. RELATED WORK

Several methods to detect the needless retransmission due to the reordering of packets have been proposed:

- The Eifel algorithm uses either the TCP time stamp option or two bits from the TCP reserved field to distinguish an original transmission from an unnecessary retransmission [9]. The Eifel algorithm is robust to up to a congestion windows worth of lost ACKs. When using the reserved bits, the algorithm requires negotiation of Eifel during the initial three-way handshake used to initiate every TCP connection.[9] does not consider varying *dupthresh* to avoid spurious retransmissions. It only backs out window reductions if the retransmission was spurious.
- The DSACK option in TCP, allows the TCP receiver to report to the sender when duplicate segments arrive at the receiver's end. Using this information, the sender can determine when a retransmission is spurious [6]. Also in their proposal, they propose storing the current congestion window before reducing the congestion window upon detection of a packet loss. Upon an arrival of a DSACK, the TCP sender can find out whether the retransmission was spurious or not. If the retransmission was spurious, then the slow start threshold is set to the previous congestion window. Their proposal does not specify any mechanisms to proactively detect reordering of packets. The main drawback of DSACK is, if an ACK containing DSACK information is dropped or corrupted by the network, the information about that particular segment is lost and the sender will never detect the spurious retransmission.

Algorithm	Description
DSACK-FA	DSACK-R + fixed FA ratio
DSACK-FAES	DSACK-FA + enhanced RTT sampling
DSACK-TA	DSACK-FA + Timeout Avoidance
DSACK-TAES	DSACK-TA + enhanced RTT sampling

TABLE I  
RR-TCP ALGORITHMS

- In [4], the authors use the DSACK information to detect whether the retransmission is spurious and propose various techniques to increase the value of *dupthresh* value. The main drawback in this proposal is that if the packets had in fact been dropped, having an increased value of *dupthresh* would not allow the dropped packets to be retransmitted quickly and the *dupthresh* value would be decreased to three DUPACKs upon a timeout.

- In [14], the authors propose mechanisms to detect and recover from false retransmits using the DSACK information. They propose several algorithms for proactively avoiding false retransmits by adaptively varying *dupthresh*. The various algorithms used are listed in Table 1.

In the DSACK-FA algorithm, the *dupthresh* value is chosen to avoid a percentage of false fast retransmit, by setting the *dupthresh* value equal to that percentile value in the reordering length cumulative distribution. The percentage of reordering the algorithm avoids is known as FA ratio.

In the DSACK-FAES algorithm, the DSACK-FA algorithm is combined with a RTT sampling algorithm which samples the RTT of retransmitted packets caused by packet delays.

The DSACK-TA algorithm uses cost functions that heuristically increase or decrease the FA ratio such that the throughput is maximized for a connection experiencing reordering. The FA ratio will increase when false retransmits occur and the FA ratio will decrease when there are significant timeouts.

In the DSACK-TAES algorithm, the DSACK-TA algorithm is combined with a RTT sampling algorithm which samples the RTT of retransmitted packets caused by packet delays.

According to [14], the DSACK-TA algorithm performed the best when compared with the other algorithms.

The methods mentioned above, with exception of [14] and [4] try to improve the TCP performance after realizing that a packet has been retransmitted due to reordering i.e these methods are reactive rather than being proactive.

### IV. EPDN: EXPLICIT PACKET DROP NOTIFICATION

When the TCP sender sends data segments to the TCP receiver through intermediate gateways, these gateways drop the incoming data packets when their queues are full or reach a threshold value. Thus the TCP sender detects congestion only after a packet gets dropped in the intermediate gateway. When a packet gets reordered in the gateway or path, the TCP sender finds it impossible to distinguish whether the data

packet has been dropped or reordered in the network. I try to solve this problem by proposing a solution to distinguish whether the packet has been dropped or reordered, by having a data structure that maintains the sequence number of the packet that gets dropped in the gateway. When an ACK for some data packet  $P_k$  arrives at the gateway, the data structure is searched to check whether the sequence number of the packet  $P_{k+1}$  has been dropped by that particular gateway or not. If the packet has been not been dropped, then a *dropped* bit is set in the ACK. When the sender receives an ACK, it checks for the *dropped* bit. Based on this information, the TCP sender can distinguish the reason for the out of order packet i.e. whether a packet has been dropped or reordered in the network and perform actions based on that decision. This mechanism of informing the TCP sender about dropped packets is called as the EPDN (Explicit Packet Drop Notification) mechanism.

1) *Data Structure used:* We use a hashtable to maintain the flow ids and the dropped packet numbers ( $PNO_i$ ) for the respective flow ids ( $F_{id}$ ). The flow id is the index and the packet numbers are the items in the list for a particular flow id.

2) *Recording information about dropped packets:*

- Initially, the hashtable is empty.
- When a packet  $\langle F_{id}, PNO_i \rangle$  gets dropped in the gateway, the corresponding flow id ( $F_{id}$ ) is used as the index to check the hashtable to find out whether there is an entry for that particular flow. If an entry is present, then sequence number of the dropped packet ( $PNO_i$ ) is inserted into the end of the list of the corresponding flow id. If an entry is not present, an entry is created, and the sequence number of the dropped packet is entered as the first entry in the list.

3) *Processing the ACK packets:* When an ACK  $\langle F_{id}, PNO_i \rangle$  arrives at the gateway,

- If the *dropped* bit is already set (some other gateway has dropped the packet), then pass on the packet.
- If the *dropped* bit is not set, the corresponding flow id ( $F_{id}$ ) is used as the index to check the hashtable. If no entry is present for that particular flow id, the *dropped* bit is not set.
- If entry is present, then the corresponding list is searched to check whether the sequence number ( $PNO_{i+1}$ ) is present. If present then the *dropped* bit is set accordingly. If the entry was not present, the *dropped* bit is not set. During the searching process, if a sequence number less than the current sequence number that is used for searching is encountered, the lesser sequence number entry is deleted from the list. This means that the packet with the lesser sequence number has been retransmitted.
- When the list is empty, the flow id entry is removed from the hashtable.

4) *Removing inactive lists:* There may be cases where possible residuals (packet sequence numbers) may be left in the list even though that particular flow has become inactive. To remove these unwanted residuals and the list for that particular flowid, we could have another hash table that maintains a timestamp of the last ACK packet that has passed through

the gateway for the flow whose entry is already present in the main hash table (When an entry is created in the main hash table for a particular flow, an entry is also created in this hashtable simultaneously). It uses the flow id ( $F_{id}$ ) as the index of the hashtable. The timestamp entry for that particular flow is regularly updated with the timestamp of the last ACK that has passed through the gateway. Regularly, say every 300 ms, the entire hashtable is scanned, and the difference of the timestamp entry in each index with the current time is calculated. If the difference is greater than a set threshold (say, 300ms), then it means that the flow is inactive currently and the entries of both the hashtables for that particular flow are removed.

## V. RD-TCP: REORDER DETECTING TCP

When the sender receives an *ack*, it checks for the *dropped* bit and if it is set, then the sender knows that the packet has been dropped and retransmits the lost packet after receiving three *dupacks*. If the *dropped* bit is not set, then the RD-TCP sender assumes that the packet has been reordered in the network and waits for 'k' more *dupacks* ('3+k' in total) instead of three *dupacks* to resend the data packet.

Assume a network with source node A and destination node B with intermediate gateways R1 and R2. Node A sends data packets  $P_1, P_2, P_3$  to node B through the gateways R1 and R2. If R1 drops packet  $P_2$  due to congestion in the network, then node B will not receive  $P_2$ . On receipt of packet  $P_3$ , node B sends a DUPACK (each having sequence number  $P_1$ ) through the gateways R2 and R1. Node A receives this DUPACK assuming the routing is purely symmetrical. When R1 drops packet  $P_2$ , the sequence number of packet  $P_2$  is inserted into the data structure at R1, which in my case is a hashtable. Node B will not receive packet  $P_2$ . When node B receives packet  $P_3$ , it sends a DUPACK. When gateway R2 receives an ACK (having sequence number  $P_1$ ), it checks whether the sequence number for packet  $P_2$  ( $P_{1+1}$ ) is available in its data structure. Since R2 does not have an entry, it does not set the *dropped* bit. When gateway R1 receives the ACK, it checks for the sequence number for packet  $P_2$ , and finds that the sequence number is present in the data structure. Gateway R2 then sets the *dropped* bit in the ACK, meaning that the packet has been dropped by the gateway. When the RD-TCP sender receives the ACK with the *dropped* bit set, the sender assumes that the packet  $P_2$  has been dropped in the gateway and retransmits the packet after receiving three DUPACKs with the *dropped* bit set.

Suppose packet  $P_2$  had been reordered in the gateway, the receiver B assuming the packet has been dropped, sends a DUPACK on receipt of packet  $P_3$ . When gateway R2 receives the ACK, it checks for the sequence number entry in its hashtable and finds that there is no entry for it, and does not set the *dropped* bit. Similarly when gateway R1 receives the ACK, it checks for the sequence number in its hashtable and finds that there is no entry for it, and does not set the *dropped* bit. When the sender node A receives a DUPACK, it checks for the *dropped* bit of each of these 2 *acks*, and when '3+k' DUPACKs with the *dropped* bit not set are received, the packet is resent and fast recovery is triggered. If the value

of 'k' is not large enough, then the sender will continue to send unnecessary retransmissions. If the value of 'k' is set too large, fast retransmit may not be triggered leading to retransmission timeout. The best value of 'k' depends on the impact of reordering and could be varied depending on the current network conditions as proposed in [4] i.e. if the sender detects spurious retransmits even though it has incremented the value of 'k', then the sender can further increase the value of 'k' to reduce the number of unnecessary retransmissions that occur due to reordering.

#### A. Avoiding false fast retransmits: Increasing dupthresh.

The RD-TCP sender assumes a packet to be reordered only when the ACK packet does not have its 'dropped' bit set. If the packets are assumed to be reordered in the network, the RD-TCP sender waits for '3+k' DUPACKs before retransmitting the packets. Thus the *dupthresh* value is increased and the retransmission procedure is delayed to avoid false fast retransmits. If the value of '3+k' is small, then the RD-TCP sender will continue to send unnecessary retransmissions. To avoid these unnecessary retransmissions, I dynamically increase the value of *dupthresh* by one (i.e. 'k' = 1) for every unnecessary retransmission i.e. if the arriving ACK packets have the 'dropped' bit not set and the current *dupthresh* value is not enough to prevent the retransmission, I increase the value of the current *dupthresh* by one.

#### B. Limited Transmit.

The limited transmit algorithm [1] allows the sender to send a new data packet for each of the DUPACKs that arrive at the sender. When the *dupthresh* value is three, the limited transmit algorithm allows the sender to send two packets beyond its current congestion window. When a greater *dupthresh* value is used, the sender is allowed to send more packets (*dupthresh* - 1 packets) beyond its current congestion window. I extend the limited transmit to send upto one additional congestion window's worth of packets when the *dupthresh* value is greater than the current congestion window. If the value is less than the current congestion window, the sender is allowed to send *dupthresh* - 1 packets.

#### C. Avoiding Timeouts: Reducing dupthresh.

There is a possibility of the sender to assume that a packet has been reordered in the network even though it had in fact been dropped. If the packets had in fact been dropped and if the *dupthresh* value is large, then the timer times out leading to retransmission of the packet and the slow start phase is entered. The sender then assumes that all contiguous packets following the dropped packet in that particular SACK block are dropped and retransmits those packets (even if the 'dropped' bit is not set) after receiving three DUPACKs (*dupthresh* value is immediately set to three).

### VI. STORAGE AND COMPUTATIONAL COSTS

The TCP options field has 40 bytes. I use one bit from the reserved bits to denote the 'dropped' bit. In my implementation

the gateways do not have to maintain the list of all the flows that pass through a particular gateway i.e. we do not maintain per-connection state for all the flows. Despite the large number of flows, a common observation found in many measurement studies is that a small percentage of flow accounts for a large percentage of the traffic. It is argued in [15] that 9% of the flows between AS pairs account for the 90% of the byte traffic between all AS pairs. It is shown in [8] that large flows could be tracked or monitored easily by using SRAM that copes up with the link speed. Our monitoring process records only flows whose packets have been dropped. To get some rough estimate of the amount of memory needed for our implementation, let us assume that there are 200,000 concurrent flows passing through one gateway, 10% of them have information about one or more dropped packets recorded in this gateway, and a non-empty list of sequence numbers of dropped packets has on average 10 entries. Thus the hash table will have 20,000 non-empty entries and the total length of the lists of sequence numbers of dropped packets will be 200,000. We need 4 bytes for each flow id, 4 bytes for each packet sequence number, and another 4 bytes for each pointer. This means that the total memory required would be about 2.5 MB. This is only a rough estimate of the amount of extra memory needed, but we believe that it is realistic. I expect an extra 8MB SRAM would be highly sufficient to implement our solution.

The computational cost mostly depends on the average length of a list of sequence numbers of dropped packets. If a flow has not dropped any packets in the gateway, then the computation done would be to check whether an entry for that particular flow id is present or not. This takes constant time computation. If a flow has a non-empty list of sequence numbers of dropped packets, then this list has to be searched whenever an ACK for that particular flow passes through the gateway. This computation takes  $O(n)$  time, if the lists are implemented in the straightforward linear way, or  $O(\log n)$  time, if the lists are implemented as suitable balanced trees ( $n$  denotes the current length of the list).

I believe that the improvement of the throughput offered by our solution justifies the extra memory and computational costs, but further investigations are needed to obtain a good estimate of the trade-off between the costs and benefits.

### VII. EPDN PERFORMANCE

In this subsection, I verify the performance of droptail gateways enabled with EPDN to droptail gateways without an EPDN mechanism. I evaluate the bottleneck link queue length of droptail gateways with and without EPDN. The simulated network has a source and destination node connected to two intermediate routers. The nodes are connected to the routers via 10 Mbps Ethernet having a delay of 1ms. The routers are connected to each other via 5 Mbps link with a delay of 50 ms. Our simulations use 1500 byte segments. I used the drop-tail queueing strategy with a queue size of 70 segments. The queue size was set such that packets were dropped when there was a buffer overflow. The experiments were conducted using a 10 long lived TCP SACK flows traversing the network topology. The maximum window size of the TCP flow was 50 segments. Each of the flows were run for 200 seconds.

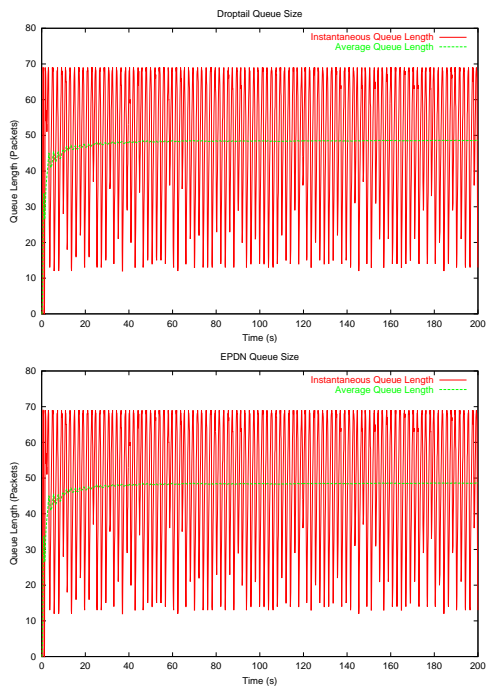


Fig. 1. Bottleneck Queue Length Comparison

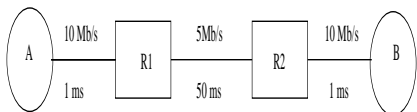


Fig. 2. Network Topology.

The graphs in Figure 1 present the plots of the instantaneous and the average queue length at the bottleneck link. Even though the EPDN mechanism has extra processing to do when packets get dropped in the gateway (inserting into the hashtable and searching the hashtable for dropped packets), it is evident from the figures that the average queue length of the droptail having EPDN is quite similar to the average queue length of the droptail without an EPDN mechanism. This proves that the extra overhead involved in an EPDN mechanism does not affect the performance of a gateway.

## VIII. RD-TCP PERFORMANCE

The simulated network has a source and destination node connected to two intermediate routers. The nodes are connected to the routers via 10Mbps Ethernet having a delay of 1ms. The routers are connected to each other via 5Mbps link with a delay of 50ms. Our simulations use 1500 byte segments. We used the drop-tail queuing strategy with a queue size of 65 segments. The experiments were conducted using a single long lived TCP flow traversing the network topology. The maximum window size of the TCP flow was 50 segments. The TCP flow lasts 1000 seconds. To introduce typical Internet link delays, we used a mean of 25 ms and standard deviation of 8 ms, such that the delay introduced varied from 0 ms to 50 ms. The 'k' value was initially set to 3 i.e.  $dupthresh = 6$ .

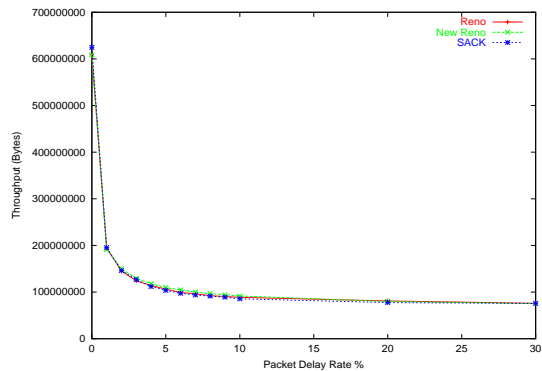


Fig. 3. Throughput versus fraction of delayed packets - Wired Network

### A. Impact of Reordering

From the Figure 3, it is evident when packets are delayed, there is a severe reduction of throughput for Reno, New Reno and SACK. This shows that persistent reordering degrades the throughput performance of a network to a large extent.

### B. Throughput: Varying Packet Delay Rate

In this section, I vary the percentage of packet delays from 1% to 30% to introduce a wide range of packet reordering events and compare the throughput performance of the simulated network using Reno, New Reno, SACK and RD-TCP.

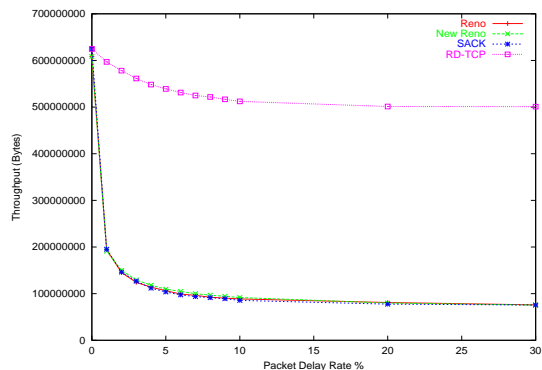


Fig. 4. Throughput versus Packet Delay Rate

As shown in the Figure 4, the throughput performance of RD-TCP is much better compared to the throughput of Reno, New Reno and SACK. For e.g. when the link experiences 1% of packet delays, RD-TCP offers a three fold throughput improvement over Reno, New Reno and SACK. The throughput of RD-TCP is almost 5 times more than Reno, New Reno and SACK when the link experiences packet delays in the range of 5% to 30%. The ability of RD-TCP to detect the packet reorder events, delay the fast retransmit procedure, prevent false fast retransmits and unnecessary reduction of the cwnd are the major reasons behind the better performance of RD-TCP over Reno, New Reno and SACK.

### C. Steady State Congestion Window

The graphs in Figure 5, present the comparison of the congestion window states of Reno versus RD-TCP, New Reno

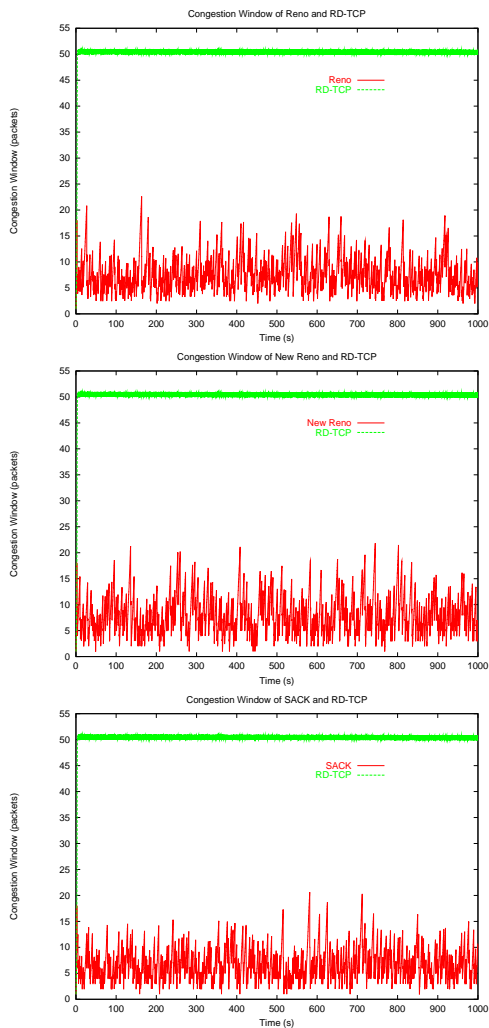


Fig. 5. Comparison of Congestion Window

versus RD-TCP and SACK versus RD-TCP when 30% of packets were delayed. When packets get delayed frequently in the network, there are unnecessary false fast retransmits causing the cwnd of Reno, New Reno and SACK to reduce very often and thus the cwnd is not able to reach the maximum window size. RD-TCP is able to distinguish a reorder event and prevent unnecessary reduction of the cwnd when packet reordering occurs. From the figures it is evident that the congestion window of Reno, New Reno and SACK could not reach the maximum window size of 50 when packets are delayed, whereas the congestion window of RD-TCP reaches the maximum window of 50 and was able to maintain a steady state throughout the experiment.

#### D. Link Utilization

The graphs in Figure 5, present the comparison of the link utilization of Reno versus RD-TCP, New Reno versus RD-TCP and SACK versus RD-TCP when 30% of packets were delayed. When packets get delayed in the network, there are unnecessary false fast retransmits causing the cwnd of Reno, New Reno and SACK to reduce by half. This causes the sender to send lesser amount of packets resulting in improper

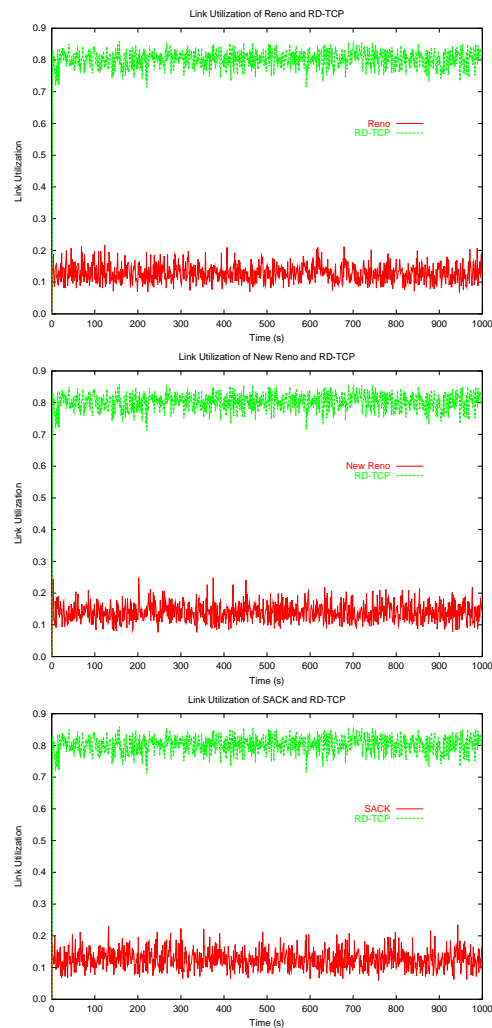


Fig. 6. Link Utilization

utilization of the available bandwidth. RD-TCP prevents unnecessary reduction of the cwnd and is able to maintain a steady sending rate. This results in proper utilization of the available bandwidth. From the figures, it is evident that RD-TCP properly utilizes the available bandwidth when compared to the improper bandwidth utilization by Reno, New Reno and SACK.

#### E. Throughput: Varying Packet Drop Rate

In this section, I compare the throughput performance of the simulated network using Reno, New Reno, SACK and RD-TCP when the link experiences both packet drops and packet delays. 5% of the packets were delayed. The packet drop rate was varied from 0% to 2%. Figure 7, reveals that the throughput of Reno, New Reno and SACK reduces considerably when more packets get dropped, whereas the throughput of RD-TCP reduces slowly compared to Reno, New Reno and SACK. When packet drops occur, the throughput of any TCP variant would reduce drastically even when there is no reordering in the network. It is to be noted that the reduction in throughput of RD-TCP is only due to packet drops and not due to false fast retransmissions caused by packet reordering.

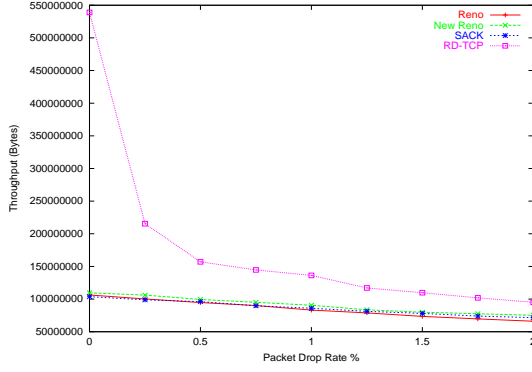


Fig. 7. Throughput versus fraction of dropped packets. 5% of packets delayed

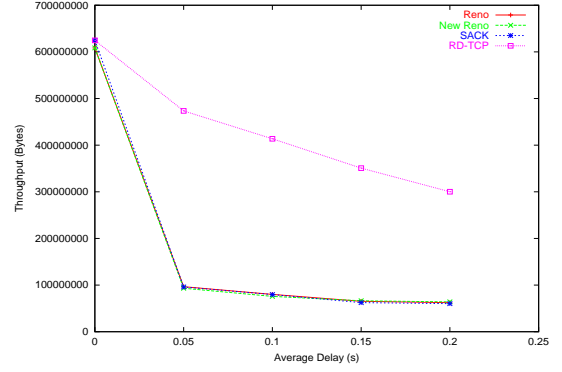


Fig. 9. Throughput versus average delay

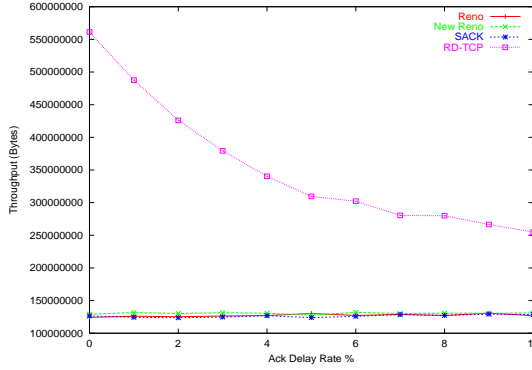


Fig. 8. Throughput versus fraction of reordered ACK packets, 3% of data packets delayed

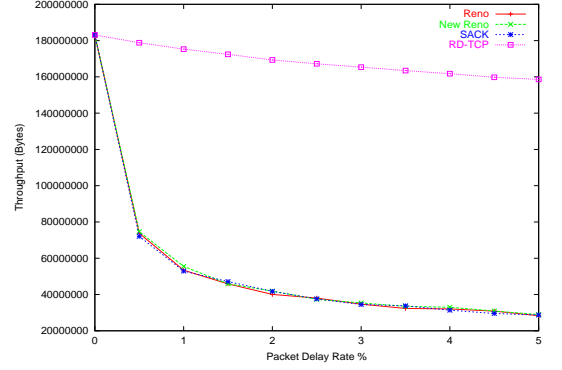


Fig. 10. Throughput versus delayed packets, 200ms propagation delay

### F. Throughput: ACK Reordering

In this section, I examine the throughput performance of the simulated network using Reno, New Reno, SACK and RD-TCP when the link experiences both data packet delays and ACK packet delays. I delayed 3% of data packets and varied the delay of ACK packets from 0% to 10%. From the Figure 8, it is evident that when reordering of ACK packets occur, the throughput of Reno, New Reno, SACK reduce negligibly. The performance of RD-TCP depends on the correct receipt of ACK packets. Even though the performance of RD-TCP reduces slowly when ACKs get reordered, RD-TCP still offers a three fold throughput performance when 5% of ACKs get reordered and a two fold throughput performance when 10% of ACKs get reordered.

### G. Throughput: Multipath Routing

In this section, I compare the throughput performance of the simulated network using Reno, New Reno, SACK and RD-TCP when multipath reordering occurs. I simulate multipath routing by introducing modal delays. This was done using modal distribution. According to [14], multipath routing produces modal delays i.e. when successive packets are sent on paths with different RTTs, these packets would be reordered proportionally to the RTT difference of the path.

When the average delay is 0.0 seconds, the packets are routed through the same path without any reordering events.

From the Figure 9, it is evident that when the average delay is increased, the throughput of Reno, New Reno and SACK reduces considerably, whereas the throughput of RD-TCP reduces much more slowly. When the average delay is 0.05 seconds, the throughput of RD-TCP is almost five times more than that of Reno, New Reno and SACK. When the average delay is increased to 0.2 seconds, RD-TCP offers a three fold improvement in throughput performance when compared to Reno, New Reno and SACK.

### H. Throughput: Large RTT

In this section, I compare the throughput performance of Reno, New Reno, SACK and RD-TCP when the link propagation delay was increased to 200 ms. The packets were also delayed from 0% to 5% using normal distribution with a mean of 100 ms and a standard deviation of 80 ms.

As shown in the Figure 10, when networks have high propagation delay, unnecessary reduction of congestion window leads to a poor throughput performance of Reno, New Reno and SACK as it could take several round trip times to achieve the maximum window size. RD-TCP's ability to distinguish packet loss from packet reordering reduces unnecessary time-outs and reduction of the congestion window thus leading to overall improvement in the throughput.

## I. Multiple Flows

In this section, I inspect an important factor to be considered while evaluating new versions of TCP - the effect of throughput performance when multiple RD-TCP flows cooperate with multiple flows of other TCP versions. I conducted a simulation on the same wired network used before but varied the number of flows that traverse the network. In order to verify the impact of reordering in the presence of multiple flows, I increased the size of the buffer to 200 segments such that there were no packet drops in the gateways. The total number of flows traversing the network were increased to ten flows, in which the sender was configured to send five Reno/New Reno/SACK flows with five RD-TCP flows. The graphs in Figure 11 present the results of the average throughput of five Reno/New Reno/SACK flows and the average throughput of five RD-TCP flows. It can be seen from the graph, that when there is no reordering the average throughput of the five Reno/New Reno/SACK flows is similar to the average throughput of the five RD-TCP flows. When the packet delay rate is increased to 0.5%, the average throughput of the other TCP flows decrease whereas the average throughput of RD-TCP increases. This does not mean the RD-TCP flows are more aggressive than the other TCP flows. Rather, it is due to the fact that the RD-TCP flows make use of the link bandwidth not utilized effectively by the Reno flows.

## J. Related Work

In this section, I compare the throughput performance of the simulated network using RN-TCP, DSACK-R and RR-TCP (we use the DSACK-TA version). I vary the percentage of packet delays from 1% to 30% to introduce a wide range of packet reordering events and compare the throughput performance of the simulated network using RR-TCP, DSACK-R and RD-TCP.

As shown in the Figure 12, the throughput performance of RD-TCP is much better compared to the throughput of DSACK-R and RR-TCP. Initially there is not much performance difference between RD-TCP and RR-TCP. When 1% to 4% of packets were delayed, RD-TCP's performance varied from 2% to 7% more than RR-TCP. When the packet delay was increased, RD-TCP's performance also increased. For e.g. when the link experiences 5% of packet delays, RD-TCP's throughput performance is 10% more than RR-TCP. When the link experiences 30% delays RD-TCP's throughput performance is 16.5% more than RR-TCP. RD-TCP also outperforms DSACK-R by offering a five fold performance improvement when the link experiences packet delays in the rate of 5% to 30%.

Figure 13 presents the throughput performance of the simulated network using RD-TCP and RR-TCP (I use the DSACK-TA version) when the link experiences both packet drops and packet delays. In particular, 30% of the packets were delayed and the packet drop rate was varied from 1% to 3%. From the figure it is evident that RD-TCP performs much better compared to RR-TCP and DSACK-R. RD-TCP's ability to distinguish packet loss from packet reordering, reduces

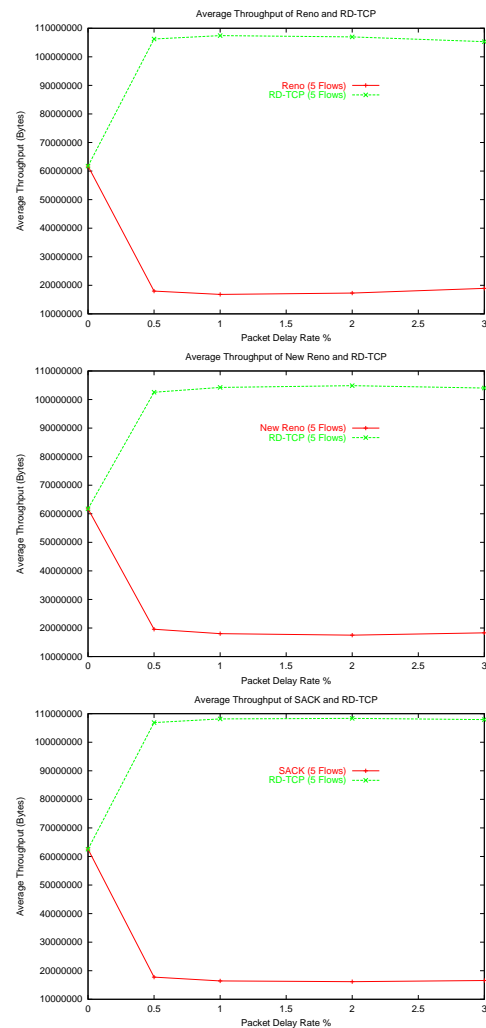


Fig. 11. Average Throughput with Multiple Flows

unnecessary timeouts and reduction of the congestion window thus leading to overall improvement in the throughput.

## IX. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a solution that prevents the unnecessary retransmits that occur due to reordering events in networks, by allowing the TCP sender to distinguish whether a packet has been lost or reordered in the network. This was done by maintaining information about dropped packets in the gateway and using this information to notify the sender, whether the packet has been dropped or reordered the gateway. Thus we proactively avoid spurious retransmits caused by reordering of packets. We also compared RD-TCP with other protocols namely TCP Reno, New Reno, SACK, DSACK and RR-TCP. We have also showed that our solution improves the throughput performance of the network to a large extent.

Further work in this area includes:

- Further work needs to be done to the TCP receiver to identify the amount of reordering that has occurred in the network and to inform the TCP sender about this

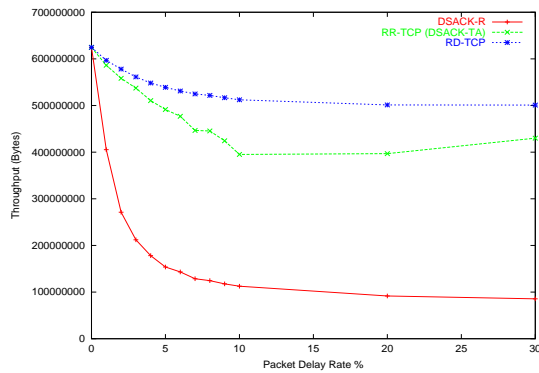


Fig. 12. Throughput versus fraction of delayed packets

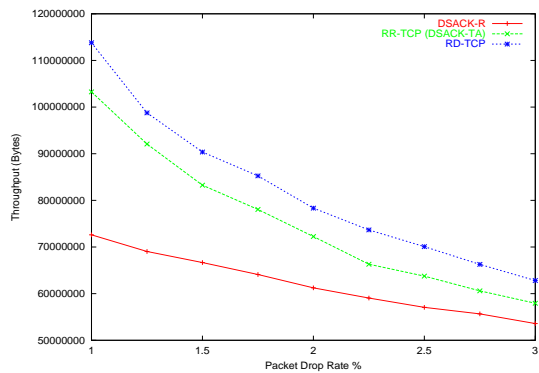


Fig. 13. Throughput versus fraction of dropped packets. 30% of packets delayed

information. The TCP sender can then increase the value of dupthresh by some value 'k' according to the degree of reordering.

- The simulated results presented in this paper needs verification in the real network.
- Further simulations and testing needs to be carried out to find the efficiency of the protocol when there is an incremental deployment i.e. when there are some routers in a network which have not been upgraded to use our mechanism.

## REFERENCES

- [1] M.Allman, H.Balakrishnan, S.Floyd, Enhancing TCP's Loss Recovery using Limited Transmit, RFC 3042, 2001.
- [2] M.Allman, V.Paxson, On Estimating End-to-End Network Path Properties. Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, Massachusetts, 1999, 263 - 274.
- [3] J.Bennett, C.Partridge, N.Shectman, Packet Reordering is Not Pathological Network Behaviour. IEEE/ACM Transactions on Networking Vol. 7, No. 6, 789 - 798, 1999.
- [4] E.Blanton, M.Allman, On Making TCP More Robust to Packet Reordering. ACM Computer Communication Review, 32(1), 2002.
- [5] B. Chinoy, ACM SIGCOMM Computer Communication Review, Volume 23, Issue 4, New York, 45 - 52, 1993.
- [6] S.Floyd, J.Mahdavi, M.Mathis, M.Podolsky, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, 2000.
- [7] V.Jacobson, Symposium proceedings on Communications architectures and protocols, California, 314 - 329, 1988.

- [8] C.Estan, G.Varghese, New Directions in Traffic Measurement and Accounting, ACM SIGCOMM 2002.
- [9] R.Ludwig, R.Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, Computer Communication Review, 30(1), 2000.
- [10] S.McCanne, S.Floyd, Network Simulator, <http://www.isi.edu/nsnam/ns/>
- [11] J.Mogul, Observing TCP Dynamics in Real Networks, ACM SIGCOMM Computer Communication Review archive Volume 22, Issue 4, 305 - 317, 1992.
- [12] J.Postel, Transmission Control Protocol, RFC 793, 1981.
- [13] A.Sathiaseelan, T. Radzik, RD-TCP: Reorder Detecting TCP, Proceedings of the 6th IEEE International Conference on High Speed Networks and Multimedia Communications HSNMC'03, Portugal, (LNCS 2720, pp.471-480), 2003.
- [14] M.Zhang, B.Karp, S.Floyd, L.Peterson, RR-TCP: A Reordering-Robust TCP with DSACK. 11th IEEE International Conference on Network Protocols (ICNP'03), Georgia, 2003.
- [15] W.Fang, L.Peterson, Inter-as-traffic patterns and their implications, IEEE GLOBECOM 1999.
- [16] V. Paxson, End-to-end routing behavior in the Internet, Proceedings of ACM SIGCOMM, Aug. 1996.
- [17] C.Ward, H. Choi, and T. Hain, A data link control protocol for LEO satellite networks providing a reliable datagram service, IEEE/ACM Transactions on Networking, 3(1):91103, Feb. 1995.