

# Reorder Notifying TCP (RN-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 decrease the TCP performance of a network, mainly because it leads to overestimation of the congestion in the network. In this paper, we analyze the performance of networks when reordering of packets occur. We propose a proactive solution that could significantly improve the performance of the network when reordering of packets occurs. We report results of our simulation experiments, which support this claim. Our solution is based on enabling the senders to distinguished between dropped packets and reordered packets.

## 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* [4]: 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 [14].
- 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 [13].
- Links with high latency: Links such as satellite links have high RTTs, typically on the order of several hundred milliseconds. Inorder to keep the pipe full, link-layer retransmission protocols send subsequent packets while awaiting an acknowledgement (ACK) or negative acknowledgement (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 [22].

The TCP (Transmission Control Protocol) is the commonly used transport protocol in the Internet. The TCP protocol provides a reliable, connection oriented, inorder delivery of data between any two hosts in the Internet [15]. To ensure the data is delivered from the sender to receiver correctly and inorder, the TCP sender uses sequence numbers to each octet

of data that is transmitted and the TCP receiver ACKs the receipt of these transmitted bytes that are received correctly and inorder. TCP has two basic methods of finding out that a segment has been lost.

### *Retransmission timer*

If an ACK for a data segment does not arrive at the sender at a certain amount of time, then the retransmission timer expires and the data segment is retransmitted [15].

### *Fast Retransmit*

When a TCP sender receives three duplicate acknowledgements (DUPACK) for a data segment  $X$ , it assumes that the data segment  $Y$  which was immediately following  $X$  has been lost, so it resends segment  $Y$  without waiting for the retransmission timer to expire [8]. Fast Retransmit uses a parameter called 'Duplicate Threshold' (*dupthresh*) which is fixed at three DUPACKs to conclude whether the network has dropped a packet.

Reordering of packets during transmission through the network has several implications on the TCP performance. The following implications are pointed out in [5]:

- 1) When a network path reorders data segments, it may cause the TCP receiver to send more than three successive DUPACKs, triggering the Fast Retransmit procedure at the TCP sender. Unnecessary retransmission of data segments means that some of the bandwidth is wasted.
- 2) The TCP transport protocol assumes congestion in the network only 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 reduces the congestion window (cwnd) to half its original size unnecessarily.

We propose extending the TCP SACK 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 some information about dropped packets and passing this information to the receiver by inserting it into the subsequent packets of the same flow. We term this mechanism of informing the receiver about dropped packets as Explicit Packet Drop Notification Version 2.0 (EPDNv2.0). Based on this information, the receiver notifies the sender whether the packet has been dropped or

reordered. We call this protocol RN-TCP (Reorder Notifying TCP). The preliminary version of this paper has been presented in [21].

Section I-A presents the previous work related to our study. Sections II, III, IV and V present the details of our proposed solution. In Sections VI, VII, VIII, IX, X, XI and XII, we describe and discuss our simulations. We conclude this paper with a summary of our work and a short discussion of the further research in Section XIII.

#### A. 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 [10]. The Eifel algorithm is robust to up to a *cwnd* 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. [10] 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 [7]. Also in their proposal, they propose storing the current *cwnd* before reducing the *cwnd* 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 *ssthresh* is set to the previous *cwnd*. 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. Throughout this paper, we term this mechanism as DSACK-R (DSACK with Recovery).
  - In [5], 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 [23], 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

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

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

- In [20], we proposed a novel method called the Explicit Packet Drop Notification Version 1.0 (EPDNv1.0) to enable the TCP senders to distinguish whether a packet has been dropped or reordered in the network by using the gateways to inform the 'sender' about the dropped packets. The gateway had to maintain information about all dropped packets for a flow, requiring considerable amount of dedicated memory at each gate. Moreover this method was proposed for networks that strictly follow symmetric routing and did not consider the case of asymmetric routing. The method proposed in the current paper overcomes both these drawbacks of the previous method. The information maintained at the gateways is substantially more concise, requiring much less memory than in the previous solution, and asymmetric routing is supported by sending the information about dropped and reordered packets to the sender via the receiver. Moreover, we have overcome the limitations of the previous method while maintaining the level of performance improvement provided by the previous method.

These methods, with exception of [23] and the method we presented in [20] are reactive and show ways of improving the TCP performance when a packet has been retransmitted in the event of reordering i.e these methods are reactive rather than being proactive. In our paper, we try to improve the performance by proactively preventing the unnecessary retransmits that occur due to the reordering event by allowing the TCP sender to distinguish whether a DUPACK received for a packet is for a dropped packet or for a reordered packet

and takes the appropriate action.

## II. OUR PROPOSED SOLUTION

We propose a solution to maintain information about dropped packets in the gateways, by having a hashtable that maintains for each flow the maximum sequence number and minimum sequence number of the packets that get dropped in the gateway. The drop can be either due to buffer overflow or due to a checksum error. When the next data packet of flow  $i$  passes through that gateway, the gateway inserts the maximum sequence number and the minimum sequence number of the dropped packets in the data packet and the entry is deleted from the data structure. We term this mechanism of explicitly informing the receiver about the dropped information as Explicit Packet Drop Notification Version 2.0 (EPDNv2.0).

We also extend TCP DSACK-R to form a new protocol called RN-TCP (Reorder Notifying TCP). The RN-TCP receiver uses the information from the EPDNv2.0 to inform the sender about whether a packet has been dropped or reordered or corrupted in the network. The RN-TCP receiver maintains two lists: the reorder list and the drop list. The elements of these lists are packet sequence numbers. When a packet is received at the receiver, the receiver uses the sequence number of the received packet, the maximum and minimum dropped information in the packet and the sequence number of the last received packet in the buffer queue to detect which packets have been dropped or reordered and inserts those sequence numbers into the drop list or the reorder list accordingly. The RN-TCP receiver uses the information present in the lists to decide whether the gap between the out-of-order packets are caused by drops or reordering and informs the RN-TCP sender about its assumption. If the packets had been dropped in the network, the RN-TCP receiver does not set the *drop-negative* bit in corresponding DUPACKs.<sup>1</sup> The RN-TCP sender then retransmits the lost packets after waiting for three DUPACKs. If the packets are assumed to be reordered in the network, the RN-TCP receiver sets the *drop-negative* bit in corresponding DUPACKs.<sup>2</sup> The RN-TCP sender then waits for '3+k' DUPACKs ( $k \geq 1$ ) before retransmitting the packets.

A detailed example on how the receiver identifies whether a data packet has been dropped or reordered is given as follows. The following case is when the current received packet  $P:n$  at the receiver is greater than the last received packet  $Q:n$  in the receiver buffer queue and  $P:n$  is greater than  $P:max$  and  $P:min$  is greater than  $Q:n$ . More details and descriptions of other cases are given in Section IV. Assume a data packet  $Q = \langle 1, 2, NULL, NULL, data_Q \rangle$  is the last received packet in the receiver buffer queue (i.e.  $Q:flowid = 1$ ,  $Q:n = 2$ ,  $Q:max = NULL$ ,  $Q:min = NULL$ ,  $Q:data = data_Q$ ). When the receiver receives the next data packet  $P = \langle 1, 7, 5, 4, data_P \rangle$  (i.e.  $P:flowid = 1$ ,  $P:n = 7$ ,  $P:max = 5$ ,  $P:min = 4$ ,  $P:data = data_P$ ), the entries  $P:max$  and  $P:min$  are checked. In this case these values are not null. If the entries are not null and if there is a gap between  $P:min = 4$

and the last received packet  $Q:n = 2$  in the receiver's buffer queue (packet 3 is missing) or a gap in between  $P:max = 5$  and the current received packet  $P:n = 7$  (packet 6 is missing), then the packets within the gap (3 and 6) have been probably reordered in the network and packets (4 and 5) have been dropped. If there was no gap, then the receiver assumes that most likely all the packets between the last received packet  $Q:n$  and the recently received packet  $P:n$  have been dropped at the gateway. If the entries  $P:max$  and  $P:min$  are empty and if there is a gap between  $P:n$  and  $Q:n$ , then the packets in between the last received packet  $Q:n$  and the current packet  $P:n$  have been probably reordered.

## III. DETAILS OF THE IMPLEMENTATION

Each gateway has a hashtable storing entries  $E = \langle flowid, timestamp, max, min \rangle$ , where  $flowid$  is the flow id (also the key used to index the hashtable),  $timestamp$  is the timestamp entry,  $max$  is the maximum sequence number and  $min$  is the minimum sequence number of the packets dropped by the gateway.  $P = \langle flowid, n, max, min, data \rangle$  denotes a data packet having entries  $flowid$ : flowid,  $n$ : sequence number,  $max$ : maximum dropped entry in the packet,  $min$ : minimum dropped entry in the packet, and  $data$ : all other data in the packet. For an entry  $E$  in a hashtable and a packet  $P$ ,  $E:field\_name$  and  $P:field\_name$  will denote the values of field  $field\_name$  in  $E$  and  $P$ , respectively.

### A. Algorithm: Recording information about dropped packets.

For each gateway, initially the hashtable is empty. When a packet  $P$  gets dropped in the gateway, the flow id  $P:flowid$  of this packet is used as the index to check the hashtable to find out whether there is an entry for this flow.

- If an entry  $E$  such that  $E:flowid = P:flowid$  is present in the hashtable (meaning that some packets of this flow have been recently dropped), then this entry is updated as follows:

$$E:max = \max\{E:max, \max\{P:n, P:max\}\},$$

$$E:min = \min\{E:min, \min\{P:n, P:min\}\}.$$

- If such an entry is not present (meaning this is the first packet of this flow to be dropped), an entry  $E$  is inserted into the hashtable, where

$$E:flowid = P:flowid,$$

$$E:max = P:n,$$

$$E:min = P:n.$$

In both cases  $E:timestamp$  is set to the current time.

### B. Algorithm: Processing the data packets at the gateway.

When a data packet  $P$  is to be sent out of the gateway and there is an entry  $E$  with  $E:flowid = P:flowid$  in the hashtable at this gateway, then the fields  $max$  and  $min$  in packet  $P$  are set as follows.

- If  $P:max$  and  $P:min$  are empty then,  $P:max = E:max$  and  $P:min = E:min$ .
- If  $P:max$  and  $P:min$  are not empty then,

<sup>1</sup>The *drop-negative* bit is 0

<sup>2</sup>The *drop-negative* bit is 1

$$P:max = \max\{E:max, P:max\}, \text{ and } P:min = \min\{E:min, P:min\}.$$

In both cases the entry  $E$  is deleted from the hashtable.

If a subsequent gateway drops the data packet carrying the dropped information, then the fields  $E:max$  and  $E:min$  would be updated accordingly. These values are then inserted into a data packet that is sent out of the gateway.

### C. 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  $F_{id}$ , we have a timestamp of the last data packet that has passed through the gateway for the flow whose entry is already present in the hashtable. The timestamp entry for that particular flow is regularly updated with the timestamp of the last data packet that has passed through the gateway. Regularly, 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, then it means that the flow is currently inactive and the entries of the hashtable for that particular flow are removed.

### D. EPDN: Implementation Issues

- Currently, for the EPDN mechanism to work successfully, all routers in the network path MUST be enabled with the EPDN mechanism. In order to verify whether all routers in the path are EPDN enabled, this requires a modification to the IP header by including a new field called the EPDN-TTL (EPDN Time To Live). Adding an EPDN-TTL field to the IP header requires another 8 bits. Initially the EPDN-TTL field is set to the value of the actual TTL (Time To Live) field. Every EPDN enabled router, decrements the EPDN-TTL along with the actual TTL while forwarding the packet. When the receiver receives the packet, the receiver checks whether  $TTL = EPDN-TTL$ . If they are equal, then that means all routers in the path are EPDN enabled. The receiver can then make a decision on the out-of-order packet. This method of detection is similar to the one proposed in [9] and is based on discussions from [17]. According to discussions from [19], many ISPs claim there's little congestion if any in the core network and we envisage the EPDN mechanism to work successfully having them deployed only in edge routers where there are cases of congestion and need not be deployed in the core network. The actual possibility of this approach is a subject of future work.
- The EPDN mechanism always inserts the maximum and minimum information into the first packet (of the same flow) that is ready to be dequeued. Currently, the EPDN mechanism will not be able to inform the RN-TCP receiver about dropped packets if there are no packets to carry that information. In that case, a packet loss will be detected only after a retransmission timeout. The sender then assumes that all out-of-order packets

following the dropped packet are dropped and retransmits those packets after receiving three DUPACKs (*dupthresh* value is immediately set to three) until the receipt of the first DSACK information meaning there has been a false fast retransmission. Incase the routing changes and if the dropped information cannot be propagated to the receiver (for instance in multipath routing), we believe the gateway could be modified to send the dropped information using out of band signalling messages to the sender analogous to source-quench [16] and BECN [3]. Possibility of this approach is a subject of future work.

- Routers accessing the TCP header: According to [19], routers can access the TCP header information using basic router commands by checking whether the upper protocol in the packet is TCP and then perform actions by accessing the sequence numbers through an offset value.

## IV. RN-TCP: IMPLEMENTATION DETAILS

When a data packet  $P$  arrives at the RN-TCP receiver, the following computation is done. The RN-TCP receiver checks whether  $P:n$  is present in the drop list or the reorder list. If present, then the entry is deleted from the list (This means that the packet  $P:n$  has arrived at the receiver and any entry with the sequence number  $P:n$  in any of these lists should be removed). The RN-TCP receiver checks whether the dropped entries  $P:min$  and  $P:max$  are empty or not.

If the dropped entries  $P:min$  and  $P:max$  are NULL, the RN-TCP receiver checks if  $P:n$  is greater than the highest received packet  $Q:n$  in the receiver buffer queue.

- If  $P:n$  is greater than  $Q:n$ , then the RN-TCP receiver checks for a gap between  $P:n$  and  $Q:n$  and if those sequence numbers required to fill the gap are present in the drop list.
  - If some of these numbers are in the drop list, then the RN-TCP receiver assumes that those packets have been dropped.
  - If not, then the packets within the gap are assumed to be reordered. The RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder list.

Example: Assume  $P:n = 6$ ,  $Q:n = 2$ ,  $P:min = NULL$  and  $P:max = NULL$ . The receiver checks whether there is a gap between packet sequence numbers 6 and 2. The receiver finds out that packets 3, 4 and 5 are missing. These sequence numbers are checked to see if they are present in the drop list. If sequence numbers 3, 4 and 5 are present in the drop list, then these packets are assumed to be dropped. Else, if sequence numbers 3, 4 and 5 are not present in the drop list, then these packets are assumed to be reordered and they are put into the reorder list.

If  $P:min$  and  $P:max$  are not NULL AND if  $P:n > Q:n$ .

- The RN-TCP receiver checks if  $P:min > Q:n$  AND  $P:max < P:n$ .

The RN-TCP receiver checks for a gap between  $P:min$  and  $Q:n$  and also checks for a gap between  $P:max$  and  $P:n$ . If there is a gap, the RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder

list. Whilst adding, check if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list. Add the sequence numbers from  $P:min$  to  $P:max$  into the drop list.

Example: Assume  $P:n = 6$ ,  $Q:n = 2$ ,  $P:min = 4$  and  $P:max = 5$ .

The receiver checks whether there is a gap between (sequence numbers 4 and 2) and (sequence numbers 5 and 6). As there is a gap (sequence number 3), the sequence number 3 is entered into the reorder list. While adding, the receiver checks if sequence numbers 4 and 5 are present in the reorder list. If they are present, those entries are deleted. The receiver then adds the sequence numbers 4 and 5 into the drop list.

- If  $P:min > Q:n$  AND  $P:max > P:n$  AND  $P:min < P:n$ . The RN-TCP receiver checks for a gap between  $P:min$  and  $Q:n$ . If there is a gap, the RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder list. Then the RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list. Put the sequence numbers from  $P:min$  to  $P:n - 1$  and  $P:n + 1$  to  $P:max$  into the drop list for future references. Example: Assume  $P:n = 6$ ,  $Q:n = 2$ ,  $P:min = 4$  and  $P:max = 7$ .

The receiver checks whether there is a gap between 4 and 2. As there is a gap (sequence number 3), the sequence number 3 is entered into the reorder list. Then the receiver checks if the sequence numbers from 4, 5, 6 and 7 are present in the reorder list. If present, the receiver removes them from the reorder list. The receiver then puts the sequence numbers 4, 5 and 7 into the drop list

- If  $P:min > Q:n$  AND  $P:min > P:n$ . Add sequence numbers from  $P:min$  to  $P:max$  into the drop list. Add sequence numbers from  $Q:n + 1$  to  $P:n - 1$  into the reorder list. Example: Assume  $P:n = 6$ ,  $Q:n = 2$ ,  $P:min = 7$  and  $P:max = 9$ . The receiver adds sequence numbers 7, 8 and 9 into the drop list and adds the sequence numbers from 3, 4 and 5 into the reorder list.
- If  $P:min < Q:n$  AND  $P:max < P:n$  AND  $P:max > Q:n$ . The RN-TCP receiver checks for a gap between  $P:max$  and  $P:n$ . If there is a gap, the RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder list. Then the RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list. Add sequence numbers from  $P:min$  to  $Q:n - 1$  and from  $Q:n + 1$  to  $P:max$  into the drop list. Example: Assume  $P:n = 6$ ,  $Q:n = 2$ ,  $P:min = 1$  and  $P:max = 4$ .

The receiver checks for a gap between 4 and 6. As there is a gap (sequence number 5), the receiver adds the sequence number 5 into the reorder list. Then the receiver checks if the sequence numbers 1, 2, 3 and 4 are present in the reorder list. If present, remove them from the reorder list. The receiver then adds the sequence

numbers 1, 3 and 4 into the drop list.

- If  $P:min < Q:n$  AND  $P:max < P:n$  AND  $P:max < Q:n$ . Add sequence numbers from  $P:min$  to  $P:max$  into the drop list. Add sequence numbers from  $Q:n + 1$  to  $P:n - 1$  into the reorder list.

Example: Assume  $P:n = 6$ ,  $Q:n = 4$ ,  $P:min = 1$  and  $P:max = 3$ .

The receiver adds sequence numbers 1, 2 and 3 into the drop list. The receiver then adds the sequence number 5 into the reorder list.

- If  $P:min < Q:n$  AND  $P:max > P:n$ . The RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list. Put the sequence numbers from  $P:min$  to  $Q:n - 1$ ,  $Q:n + 1$  to  $P:n - 1$  and  $P:n + 1$  to  $P:max$  into the drop list for future references.

Example: Assume  $P:n = 6$ ,  $Q:n = 4$ ,  $P:min = 1$  and  $P:max = 7$ .

The receiver checks if the sequence numbers 1, 2, 3, 4, 5, 6 and 7 are present in the reorder list. If present, remove the sequence numbers from the list. The receiver then puts the sequence numbers 1, 2, 3, 5 and 7 into the drop list.

If  $P:min$  and  $P:max$  are not NULL AND if  $P:n < Q:n$ .

- If  $P:min > Q:n$ . The RN-TCP receiver checks for a gap between  $P:min$  and  $Q:n$ . If there is a gap, the RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder list. Put the sequence numbers from  $P:min$  to  $P:max$  into the drop list for future references. Example: Assume  $P:n = 5$ ,  $Q:n = 9$ ,  $P:min = 11$  and  $P:max = 12$ . The receiver checks for a gap between 11 and 9. As there is a gap (sequence number 10), the receiver adds the sequence number 10 into the reorder list. The receiver then puts the sequence numbers 11 and 12 into the drop list for future references.
- If  $P:min < P:n$  AND  $P:max > P:n$  AND  $P:max > Q:n$ . The RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list. Put the sequence numbers from  $P:min$  to  $P:n - 1$ ,  $P:n + 1$  to  $Q:n - 1$  and  $Q:n + 1$  to  $P:max$  into the drop list for future references. Example: Assume  $P:n = 5$ ,  $Q:n = 9$ ,  $P:min = 4$  and  $P:max = 10$ . The receiver checks if the sequence numbers 4, 5, 6, 7, 8, 9 and 10 are present in the reorder list. If present, remove them from the reorder list. The receiver then puts the sequence numbers 4, 6, 7, 8 and 10 into the drop list.
- If  $P:max < Q:n$  AND  $P:max > P:n$  AND  $P:min < P:n$ . The RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $Q:n$  are present in the reorder list. If present, remove them from the reorder list. Put the sequence numbers from  $P:min$  to  $P:n - 1$  and  $P:n + 1$  to  $P:max$  into the drop list for future references.

Example: Assume  $P:n = 5$ ,  $Q:n = 9$ ,  $P:min = 4$  and  $P:max = 8$ .

The receiver checks if the sequence numbers 4, 5, 6, 7, 8 and 9 are present in the reorder list. If present, remove them from the reorder list. The receiver then puts the sequence numbers 4, 6, 7 and 8 into the drop list.

- If  $Q:n > P:max$  AND  $P:min > P:n$ .

Add sequence numbers from  $P:min$  to  $P:max$  into the drop list.

Example: Assume  $P:n = 5$ ,  $Q:n = 9$ ,  $P:min = 6$  and  $P:max = 8$ .

The receiver adds the sequence numbers 6, 7 and 8 into the drop list.

- If  $P:min < Q:n$  AND  $P:max < P:n$ .

The RN-TCP receiver checks for a gap between  $P:max$  and  $P:n$ . If there is a gap, the RN-TCP receiver adds those sequence numbers required to fill the gap to the reorder list. The RN-TCP receiver checks if the sequence numbers from  $P:min$  to  $P:max$  are present in the reorder list. If present, remove them from the reorder list and add them to the drop list.

Example: Assume  $P:n = 5$ ,  $Q:n = 9$ ,  $P:min = 2$  and  $P:max = 3$ .

The receiver checks for a gap between sequence numbers 3 and 5. As there is a gap (sequence number 4), the receiver adds the sequence number 4 into the reorder list. The receiver then checks if the sequence numbers 2 and 3 are present in the reorder list. If the sequence numbers are present, they are removed from the reorder list and added into the drop list.

#### A. RN-TCP receiver algorithm: Sending acknowledgements.

When the received data packet has been processed, the TCP receiver does the following,

- If an incoming packet  $P:n$  is filling a gap such that there are no out-of-order packets in the receiver queue, then check if packet  $Q:n + 1$  is in the reorder list. If yes, the packet  $Q:n + 1$  is assumed to be reordered and the *drop-negative* bit is set and the cumulative ACK is sent. Otherwise the packet  $Q:n + 1$  is assumed to be dropped and the cumulative ACK is sent without setting the *drop-negative* bit.
- If the packet does not fill a gap, then the receiver checks whether the sequence number following the last in-order packet is in the reorder list. If yes, the packet following the last in-order packet is assumed to be reordered and the *drop-negative* bit is set for that particular SACK packet. Otherwise the packet following the last in-order packet is assumed to be dropped and the SACK is sent without setting the *drop-negative* bit.

#### B. Sender side: Implementation Details

1) *Limited Transmit.*: When the *dupthresh* value is three, the limited transmit algorithm allows the sender to send two packets beyond its current *cwnd*. Generally, when a greater *dupthresh* value is used, the sender is allowed to send *dupthresh* - 1 packets beyond its current *cwnd*. We extend the

limited transmit to send up to one additional *cwnd*'s worth of packets when the *dupthresh* value is greater than the current *cwnd*. If the value is less than the current *cwnd*, the sender is allowed to send *dupthresh* - 1 packets.

2) *Avoiding false fast retransmits: Increasing dupthresh.*: The TCP sender assumes a packet to be reordered only when the ACK packet has the *drop-negative* bit set and the ECE bit is set to zero (i.e. No ECN information has been received recently). If the packets are assumed to be reordered in the network, the TCP sender waits for more than three DUPACKs before retransmitting the packets i.e. the *dupthresh* value is increased and the retransmission procedure is delayed to avoid false fast retransmits and unnecessary *cwnd* reductions. The *dupthresh* value is calculated as follows:

$$dupthresh = \max(4, swnd + (swnd \times kval))$$

where *swnd* is the size of the previously sent window of packets (minimum of *cwnd* and the *rwnd*) and *kval* is a constant which is set based on the RTT in the following way.

$$kval = \max(\lceil \frac{AverageRTT}{AbsoluteRTT} \rceil \times c, kval)$$

where AverageRTT is the connection's smoothed RTT average, AbsoluteRTT is the minimum observed SampleRTT value and *c* is a constant set to 0.1. The value of *c* was determined to give the best results after heuristically testing for different values of *c*. Initially, the value of *kval* is 0. The idea of increasing *dupthresh* in terms of *swnd* is to ensure that the sender waits for the previously sent packets to reach the receiver assuming that the packet could have been reordered up to a distance of the previously sent window (*swnd*) of packets. The RTT is used as a multiplicative factor to ensure that the sender waits longer for larger delays. This is useful in satellite networks where the RTT is large and where packets could have been reordered in multiples of RTT due to link level retransmissions. If the value of *kval* is not sufficient enough to prevent a false fast retransmission, then the value of *kval* is doubled for each false fast retransmission.

3) *Reacting to packet drops.*: If any of the three duplicate ACKs received have their *drop-negative* bit un-set, then the RN-TCP sender assumes that the packet has been dropped in the network. So the sender retransmits the lost packet after receiving three duplicate ACKs and enters fast recovery.

4) *Avoiding Timeouts: Reducing dupthresh.*: There is a possibility of packets being dropped in the gateways while the receiver assumes these packets have been reordered (for instance the routing changes and if the dropped information cannot be propagated to the receiver, for example in the case of multipath routing) and sets the *drop-negative* bit in corresponding duplicate ACKs. 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 value of *kval* is set to 0. The sender then assumes that all out-of-order packets following the dropped packet are dropped and retransmits those packets (even if the *drop-negative* bit is set) after receiving three duplicate ACKs (*dupthresh* value is immediately set to three) until the receipt of the first DSACK information.

## V. STORAGE AND COMPUTATIONAL COSTS

The IP option field has 40 bytes of unused space. We use the IP options to store the maximum and minimum dropped information in the packet. We use 4 bytes for each minimum and maximum dropped entries to be inserted into the option field of the IP header. We need another byte for representing the EPDN-TTL field. We use one bit from the reserved bits from the TCP header to denote the *drop-negative* bit. In our implementation we 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. Our monitoring process records only flows whose packets have been dropped. When the dropped information is inserted into the corresponding packet that leaves the gateway successfully, the entry is deleted. Thus, the gateway maintains only limited information in the hash table. 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 the gateway, 10% of them have information about one or more dropped packets recorded in this gateway. Thus the hash table will have 20,000 flow-id entries with 2 entries corresponding to the maximum and minimum dropped sequence numbers. We need 4 bytes for each flow-id, 4 bytes for the timestamp entry, 4 bytes for each packet sequence number, and another 4 bytes for each pointer. This means that the total memory required would be about 560 KB. This is only a rough estimate of the amount of extra memory needed, but we believe that it is realistic. Thus we expect that an extra 2MB SRAM would be highly sufficient to implement our solution.

The computational costs in the gateways are mostly constant time. 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 dropped packets, then inserting the information into the packet takes constant time. Deleting the entry also takes constant time. The computational costs at the receiver are as follows: The cost of maintaining the reorder list and drop list depends on the number of packets the TCP receiver assumes that have been reordered and dropped in the network. Thus the computational cost involved in Insertion is  $O(n)$  where  $n$  is the number of packets the receiver has assumed to be dropped or reordered. Deletion and comparison cost  $O(m)$  where  $m$  is the length of the list. The computational cost can be  $O(\log n)$  and  $O(\log m)$  respectively if we use balanced trees.

We believe that the throughput improvement 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.

## VI. TOPOLOGY

The simulated network (figure VI) 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 1 ms. The routers are connected to each other via link with variable link capacity and variable delay. Our simulations use 1500 byte segments. We used the drop-tail queuing strategy

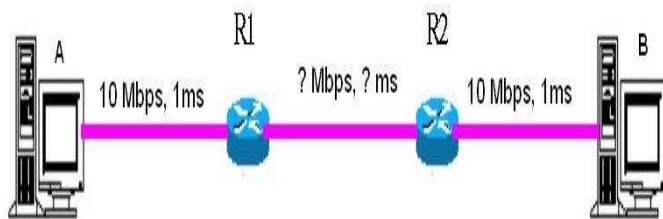


Fig. 1. RN-TCP:Simulation Scenario

with a queue size set to the bandwidth delay product. The experiments were conducted using a one or more long lived FTP flows traversing the network topology. The maximum window size of the TCP flow was also set to the bandwidth delay product. The TCP flow lasts 1000 seconds. We compared the performance of RN-TCP with SACK, DSACK-R and RR-TCP(DSACK-TA).

### A. Simulating Reordering

Reordering is caused when a delayed packet with a higher sequence number is scheduled to traverse the link later than an un-delayed packet with a lower sequence number. In order to simulate packet reordering, we delay a percentage of packets traversing the link by delay distributions. We classify the reordering distributions into mild delay distribution, modal delay distribution and large delay distribution.

- Parallelism in routers (software or hardware) result in relatively mild reordering lengths.
- 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.
- Satellite links have long propagation delays. The packets that get delayed due to the retransmission at the link layer experience delays by multiples of the RTT.

We simulate a wide variety of delay distributions such as Normal, Uniform and Exponential distributions to demonstrate the value of our algorithms in improving the performance under reordering.

## VII. MILD REORDERING

In this section, we verify the performance of RN-TCP when packets undergo relatively mild reordering lengths. The link between the routers was set to 6 Mbps capacity with a propagation delay of 50 ms. The process of delaying a packet was normally distributed with a mean packet delay of 25 ms and standard deviation of 8 ms, such that the delay introduced varied from 0 ms to 50 ms. The parameters representing propagation delay and packet delay process represents typical Internet link delays and relatively mild reordering.

### A. Throughput: Varying Packet Delay Rate

In this section, we 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 TCP SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP.

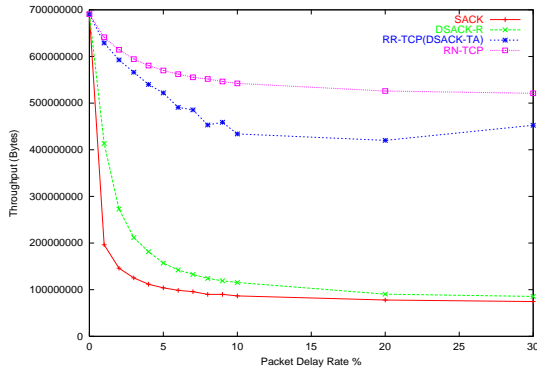


Fig. 2. Mild Delay Distribution - Throughput versus fraction of delayed packets.

As shown in the Figure 2, the throughput performance of RN-TCP is much better compared to the throughput of TCP SACK, DSACK-R and RR-TCP(DSACK-TA) for all tested packet delay rates. For example, when the link experiences 5% of packet delays, RN-TCP’s throughput performance is almost 9% more than RR-TCP(DSACK-TA), 4 times more than DSACK-R and almost 5 times more than SACK. When the link experiences 10% of packet delays, RN-TCP’s throughput performance is almost 25% more than RR-TCP(DSACK-TA), 5 times more than DSACK-R and almost 6 times more than SACK.

As shown in the Figure 3, when the packet delay rate increases, RN-TCP does not undergo any unnecessary *cwnd* reductions due to false fast retransmissions followed by RR-TCP(DSACK-TA), for which the number of times the *cwnd* was reduced due to fast retransmission was less than 0.3%. SACK and DSACK-R undergo large number of unnecessary *cwnd* reductions.

The ability of RN-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 RN-TCP over SACK, DSACK-R and RR-TCP(DSACK-TA).

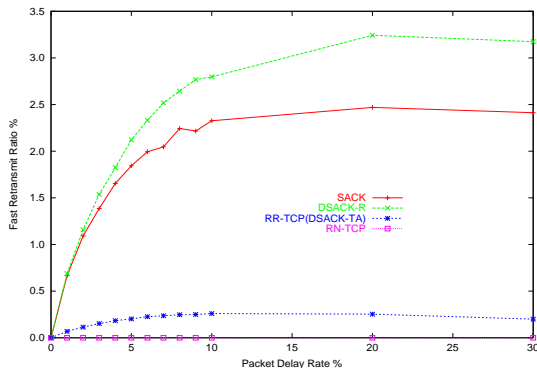


Fig. 3. Mild Delay Distribution - Fast retransmit ratio versus fraction of delayed packets.

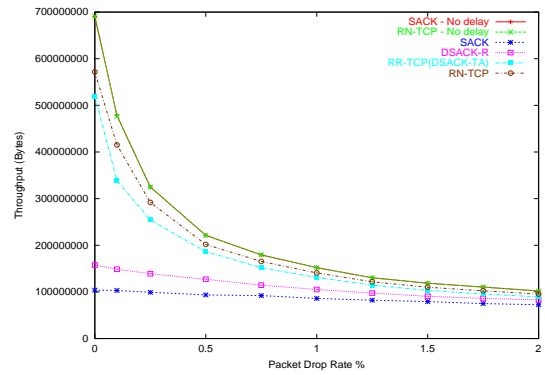


Fig. 4. Mild Delay Distribution - Throughput versus fraction of dropped packets. 5% of packets delayed.

### B. Throughput: Varying Packet Drop Rate

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when the link experiences both packet drops and packet delays. 5% of the packets were delayed. We also compared the performance of SACK and RN-TCP with packet drops only. The packet drop rate varied from 0% to 2%. Figure 4, reveals that the throughput of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP reduce considerably when packets get dropped. When packet drops occur, the throughput of any TCP variant would reduce drastically even when there is no reordering in the network. This is evident from the graph, where the performance of SACK with no delay reduces drastically with increasing packet drops. Moreover, our RN-TCP with no delay performs similar to SACK with no delay. Thus, it is clear that given there are no reordering events, RN-TCP performs similar to SACK. Unlike RR-TCP(DSACK-TA), which can only identify a false fast retransmit when there are no packet loss within that window of packets, RN-TCP can identify whether a packet has been reordered or dropped even when there are packet losses within that window of packets. From the Figure it is evident that RN-TCP’s performance is slightly better compared to SACK, DSACK-R and RR-TCP(DSACK-TA). For large packet drop rates, the throughput of all protocols are almost similar and the effect of RN-TCP and RR-TCP(DSACK-TA) that improve the performance in the presence of reordering is negligible.

It is also evident from Figure 5, that the number of timeout events of RN-TCP is much less compared to RR-TCP(DSACK-TA). From the Figure 6, it is clear that RR-TCP(DSACK-TA) does not have the ability to detect whether a packet has been dropped or reordered instantaneously and thus has a large *dupthresh* value, leading to more timeout events. RN-TCP is able to distinguish packet loss from packet reordering and retransmits the dropped packet after receiving three DUPACKs. This reduces unnecessary timeout events. In case a packet drop is detected after a timeout event, RN-TCP sender then assumes that all out-of-order packets following the dropped packet are dropped until the receipt of the first DSACK information and retransmits those packets (even if the ‘drop-negative’ bit is set) after receiving three DUPACKs

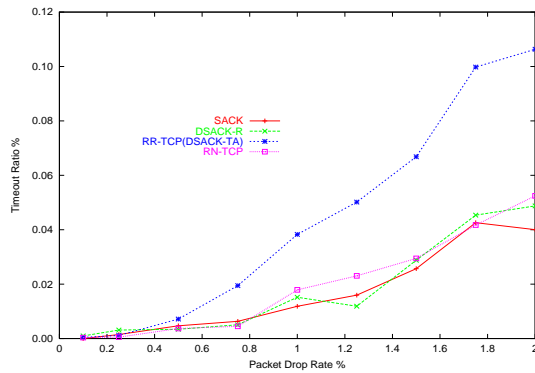


Fig. 5. Mild Delay Distribution - Timeout ratio versus fraction of dropped packets. 5% of packets delayed.

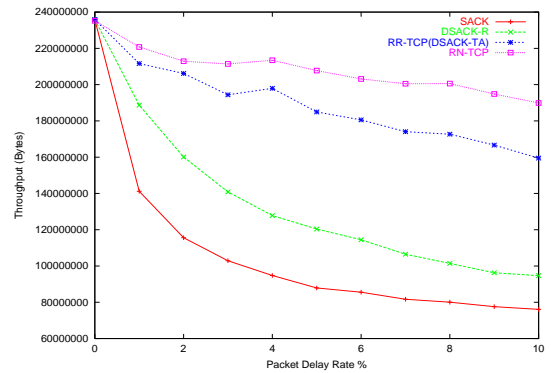


Fig. 7. Mild Delay Distribution - Throughput versus fraction of delayed packets, with bursty packet loss.

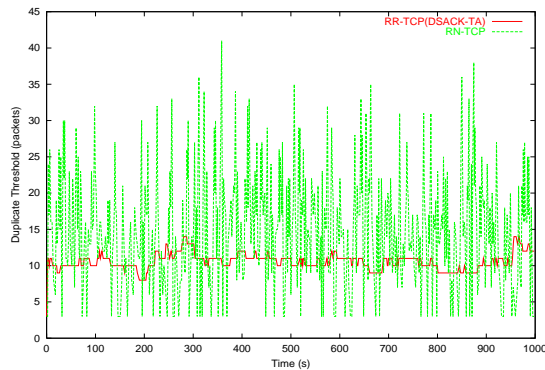


Fig. 6. Mild Delay Distribution - Dupthresh value versus time.

(*dupthresh* value is immediately set to three). When RN-TCP assumes that the packet has been reordered, it immediately transits to a higher *dupthresh* value to prevent false retransmissions.

### C. Throughput: Presence of Bursty Packet Loss

In this section, we verify the performance of the protocols under reordering and bursty packet loss, when multiple packets are lost within a window of data. Bursty packet loss occurs under low statistical multiplexing, where one or more connections in slow start cause a *drop-tail* router to drop a burst of packets. In order to verify the performance, we drop 0.4% of packets that traverse the bottleneck link throughout the simulation period except during  $[300, 400]ms$  where we drop 10% of the packets that traverse the bottleneck link. This causes bursty packet loss behavior such that there are multiple packet drops within a window of data, many of which are consecutive packets. We also vary the packet delay rate from 0 to 10%. From Figure 7, it is evident that RN-TCP performs better compared to the other protocols. For example, when the packet delay rate is 5%, RN-TCP gives a 12% throughput improvement over RR-TCP(DSACK-TA), a 73% throughput improvement over DSACK-R and a two fold throughput improvement over SACK. Similarly when the packet delay rate is increased to 10%, RN-TCP gives a 19% throughput improvement over RR-TCP(DSACK-TA) and

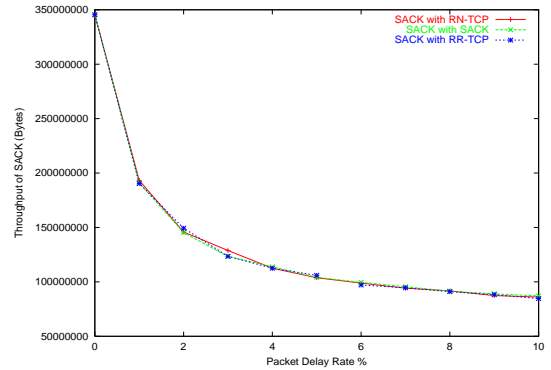


Fig. 8. Mild Delay Distribution - Throughput of SACK while co-operating with RN-TCP and RR-TCP(DSACK-TA) versus fraction of reordered packets

gives a two fold throughput improvement over DSACK-R and SACK. When there is no packet delay, all TCP variants give the same throughput performance.

### D. Fairness

When a protocol gives an throughput performance improvement over a standard protocol, the question of fairness arises. According to [23], the poor throughput performance of SACK is due to the reordering of packets that occur on the network. The enhanced versions such as DSACK-R, RR-TCP(DSACK-TA) or RN-TCP does not cause reordering to occur and thus not responsible for the poor throughput performance of SACK. For instance, when SACK competes with an enhanced version on a path that reorders packets, replacing the enhanced version with a SACK will not improve the throughput performance of the other competing SACK. In order to verify this, we examine the performance of a single flow using SACK when it competes with a single flow using RN-TCP, a single flow using SACK when it competes with a single flow using RR-TCP(DSACK-TA) and when a single flow using SACK competes with a single flow using SACK. We varied the delay of data packets from 1% to 10% using a mean packet delay of 25 ms and a standard deviation of 8 ms such that the packets chosen for delay varies from 0 to 50 ms. Moreover, in order to analyze the fairness when reordering happens, we ensure

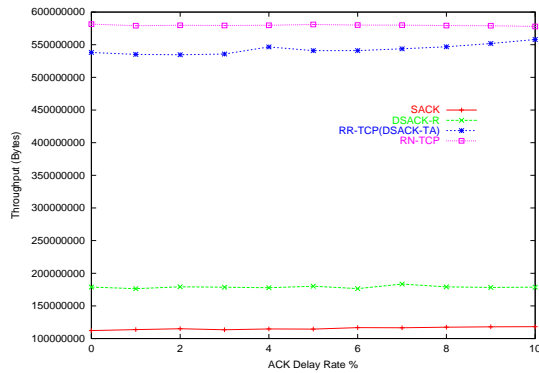


Fig. 9. Mild Delay Distribution - Throughput versus fraction of reordered ACK packets, 4% of data packets delayed.

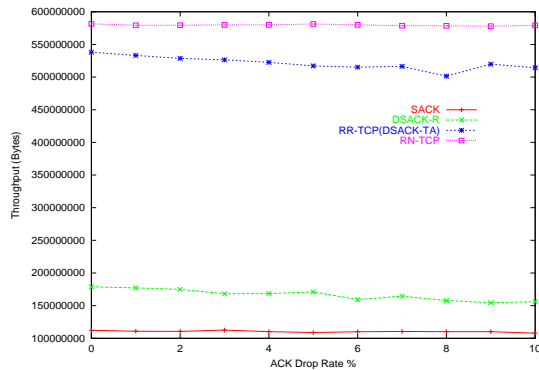


Fig. 10. Mild Delay Distribution - Throughput versus fraction of dropped ACK packets, 4% of data packets delayed.

that packets do not get dropped by having a suitable queue size. From the Figure 8, it is clear that when there are pure reordering events, an enhanced version of SACK does not cause SACK to perform badly. This is evident from the graph, where the performance of SACK with RR-TCP(DSACK-TA) and SACK with RN-TCP perform almost similar to SACK with SACK.

#### E. Throughput: ACK Reordering

Reordering could occur on the reverse path, such that the sender receives out-of-order ACKs. In order to analyze the effect of reordered ACKs on the sender's throughput performance, we examine the performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when the link experiences both data packet delays and ACK packet delays. We delayed 4% of data packets and varied the rate of delayed ACK packets from 1% to 10%. From the Figure 9, it is evident that reordering of ACK packets does not have any significant impact on the the throughput performance of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP.

#### F. Throughput: ACK Drops

We would also like to verify whether loss of ACK packets have any effect on the throughput performance of the

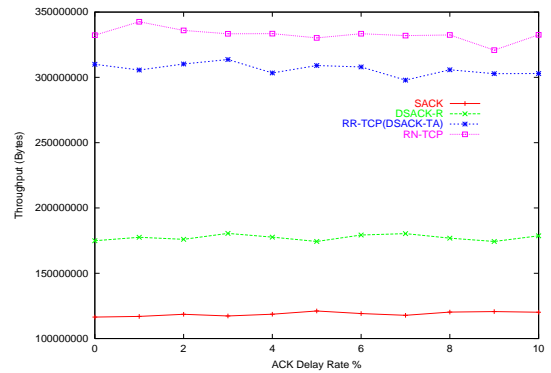


Fig. 11. Mild Delay Distribution - Throughput versus fraction of reordered ACK packets, 5% of data packets delayed, 0.2% of data packets dropped, 0.2% of ACK packets dropped.

sender. In this section, we examine the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when the link experiences both data packet delays and ACK packet drops. We delayed 4% of data packets. 1% to 10% of the ACK packets were dropped. From the Figure 10, it is evident that when ACK packets get dropped, there is no significant impact on the throughput of all the protocols.

#### G. Throughput: Dropped and Reordered Data and ACK packets

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when the link experiences packet drops, packet delays, ACK drops and ACK delays. We delayed 5% of data packets, dropped 0.2% of data packets, dropped 0.2% of ACK packets and varied the rate of delayed ACK packets from 1% to 10%. From the Figure 11, it is evident that RN-TCP outperforms the other protocols for all tested ACK delay rates. Moreover delayed and dropped ACK packets do not have any significant impact on the the throughput performance of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP.

#### H. Throughput: Various Reordering Distribution

The graphs in Figure 12, present the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP for various reordering distributions. We compared the performance when the packet delay process undergoes normal, exponential and uniform distributions. The packet delay process varied from 0 ms to 50 ms. The normal distribution was configured with a mean packet delay of 25 ms and a standard deviation of 8 ms. The uniform distribution was configured between 0 ms and 50 ms. The exponential distribution was configured with a mean of 25 ms. It is evident from the Figure that RN-TCP outperforms SACK, DSACK-R and RR-TCP(DSACK-TA) independent of any reordering distribution.

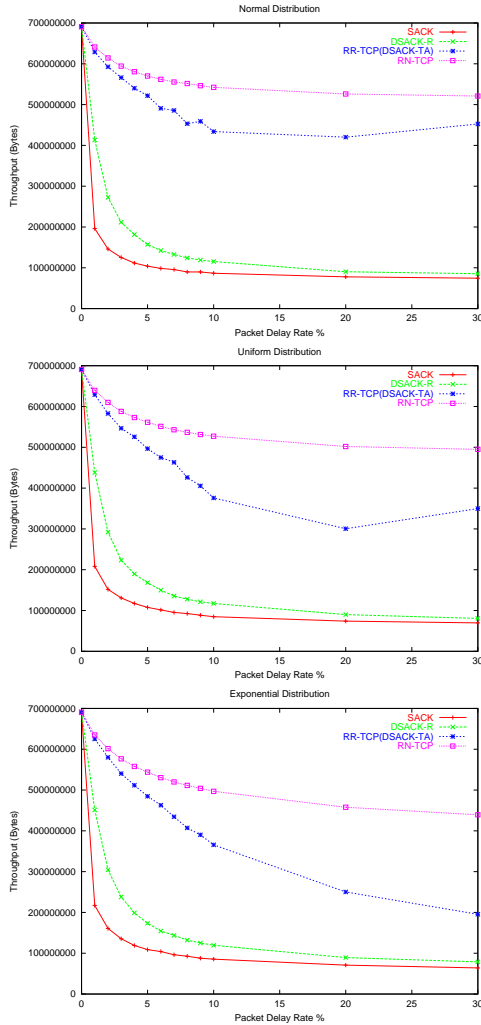


Fig. 12. Various probability distributions.

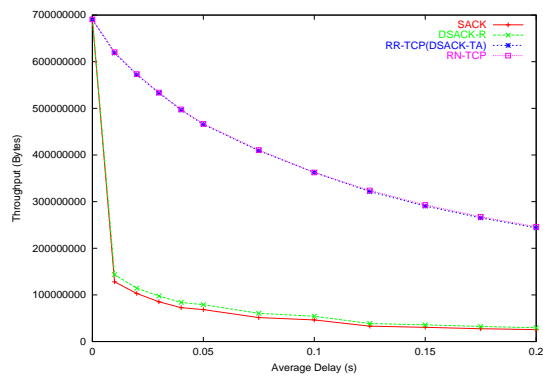


Fig. 13. Modal Delay Distribution - Throughput versus average packet delay.

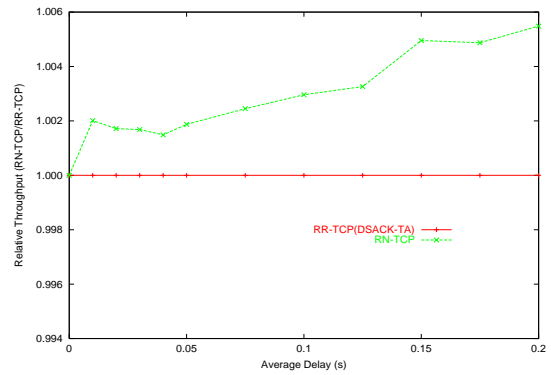


Fig. 14. Modal Delay Distribution - Relative throughput versus average packet delay.

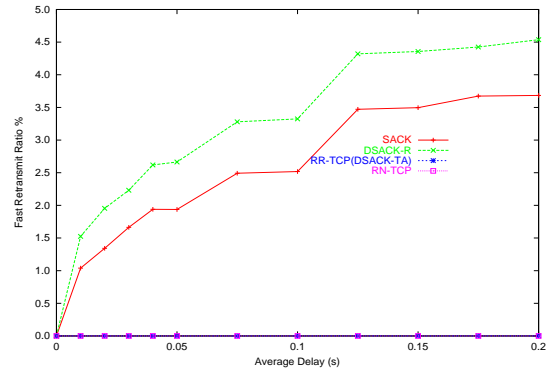


Fig. 15. Modal Delay Distribution - Fast Retransmit Ratio versus average packet delay.

### VIII. MULTIPATH ROUTING

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when multipath reordering occurs. We assume that the sender's packets were sent alternatively over two paths having different RTTs. Each of these paths have a fixed RTT. The link between the routers was set to 6 Mbps capacity with a delay of 50 ms. We varied the average packet delay from 0.0 seconds to 0.2 seconds (upto  $2 \times RTT$ ). The average packet delay represents the RTT difference between the 100 ms RTT path and the longer path. When the average packet delay is 0.0 seconds, the packets are routed through the same path without any reordering events. From the Figure 13, it is evident that when the average packet delay is increased, the throughput of SACK and DSACK-R reduce considerably, whereas the throughput of RR-TCP(DSACK-TA) and RN-TCP reduce much more slowly. The throughput performance decrease of RN-TCP is due to the idle time caused by the limited transmit algorithm, as we have restricted the limited transmit to send one window's worth of packets. This was done to prevent TCP from delaying its response to a genuine packet loss. For all tested average packet delays, RN-TCP outperforms SACK and DSACK-R. Figure 14 presents the relative throughput of RN-TCP over RR-TCP(DSACK-TA). From the Figure, it is clear that for all tested average packet delays RN-TCP achieves almost similar

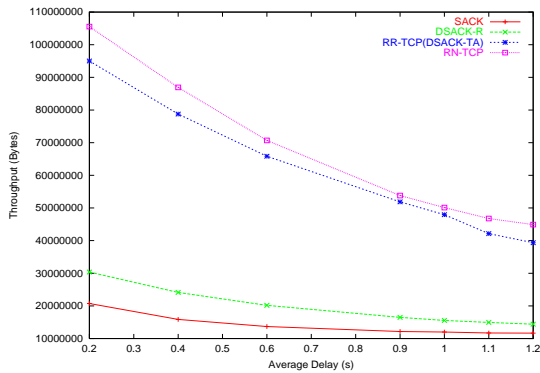


Fig. 16. Large Delay Distribution - Throughput versus average packet delay.

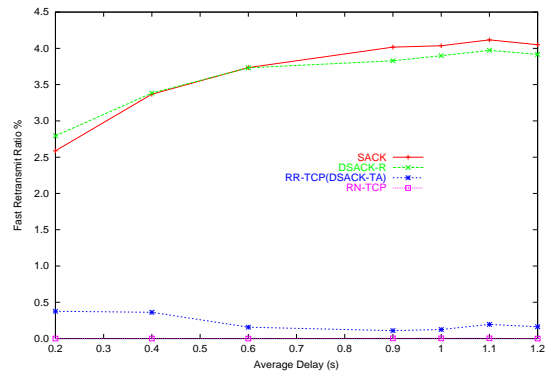


Fig. 18. Fast retransmit ratio versus average packet delay.

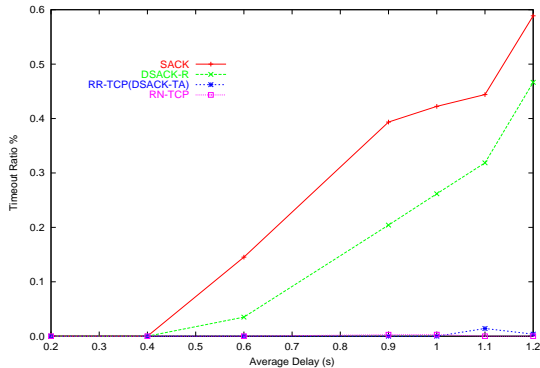


Fig. 17. Timeout ratio versus average packet delay.

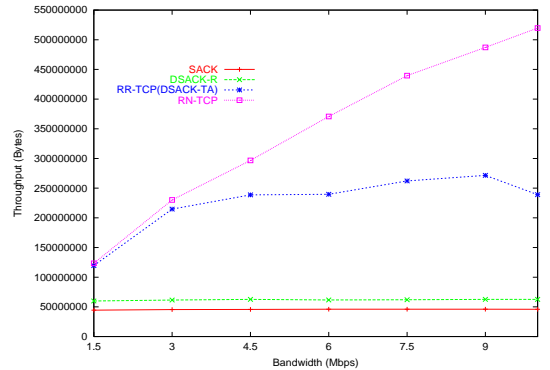


Fig. 19. Throughput versus varying bandwidth, 80 ms propagation delay.

throughput to RR-TCP(DSACK-TA). As the average delay distribution does not vary during the simulation period, both RN-TCP and RR-TCP(DSACK-TA) increase their *dupthresh* values rapidly such that no false retransmissions occur and stay with that value for the entire simulation period. The *dupthresh* value grows more rapidly when compared to RR-TCP(DSACK-TA), causing a slight throughput improvement. From the Figure 15, it is evident that both RN-TCP and RR-TCP(DSACK-TA) have almost a zero fast retransmit ratio compared to DSACK-R and SACK.

## IX. SEVERE REORDERING

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when packets experience large delay distributions in the order of multiples of RTT. The link between the routers was set to 1.5 Mbps capacity with a propagation delay of 200 ms. This propagation delay represents an upper range of Internet link delays which are predominant in satellite networks. To introduce severe packet delays, we used a mean packet delay of  $yP$  s ( $P$  is the propagation delay) and standard deviation of  $\frac{y}{3}P$  s, such that the delay introduced varied from 0 and  $2yP$  s. The packet delay rate was fixed at 7%. We varied the value of  $y$  from 1.0 to 6.0. From the Figure 16, when packets experience an average packet delay of 0.2 s, the throughput of RN-TCP is 14% more than the throughput of RR-TCP(DSACK-TA),

almost five times more than SACK and four times more than DSACK-R. When packets experience an average packet delay of 1.2 s, the throughput of RN-TCP is 18% more than RR-TCP(DSACK-TA), four times more than SACK and three times more than DSACK-R.

Moreover, RN-TCP has a near zero timeout ratio for all tested average packet delays whereas RR-TCP(DSACK-TA) experiences more timeout events when the average packet delay is more than 1.0 s (see Figure 17). Both SACK and DSACK-R experiences more timeouts when the average packet delay is increased. RN-TCP has a zero fast retransmit ratio when compared to RR-TCP(DSACK-TA) which has a slightly higher fast retransmit ratio. SACK and DSACK-R undergo large number of *cwnd* reductions due to fast retransmits (see Figure 18).

## X. VARYING BANDWIDTH WITH CONSTANT DELAY

In this section, we compare the throughput performance of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP with packet reordering events for various link capacities varying from [1.5,10] Mbps. The delay was fixed at 80 ms. 10% of packets were delayed using uniform distribution  $[0, 2P]$ , where  $P$  is the propagation delay.

As shown in the Figure 19, when the link capacity is set to 3 Mbps, the throughput of RN-TCP is almost 7% more than RR-TCP(DSACK-TA), five times more than SACK and three times more than DSACK-R. When the link capacity is set

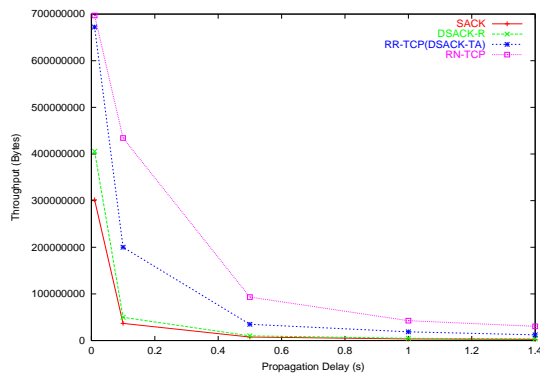


Fig. 20. Throughput versus varying delay, 10 Mbps bandwidth.

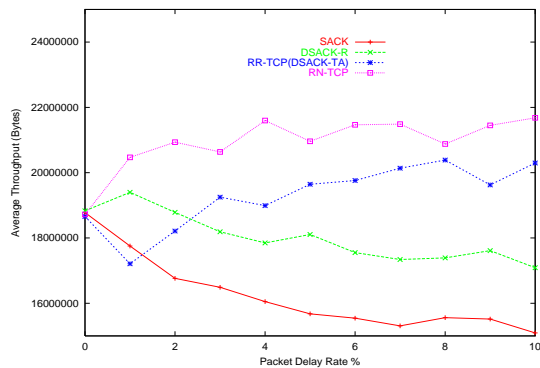


Fig. 21. Mild Delay Distribution - Average Throughput versus packet delay rate.

to 10 Mbps, the throughput of RN-TCP is twice more than RR-TCP(DSACK-TA), eleven times more than SACK and eight times more than DSACK-R. Thus it is evident that RN-TCP outperforms SACK, DSACK-R and RR-TCP(DSACK-TA) irrespective of the link capacity.

## XI. VARYING DELAY WITH CONSTANT BANDWIDTH

In this section, we compare the throughput performance of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP when the propagation delay varied from [10 ms, 1.4 s]. The bandwidth was fixed at 10 Mbps. 10% of packets were delayed using uniform distribution  $[0, 2P]$ , where  $P$  is the propagation delay.

As shown in the Figure 20, when the link delay is set to 0.01 s, the throughput of RN-TCP is almost 4% more than RR-TCP(DSACK-TA), twice more than SACK and DSACK-R. When the link delay is set to 1.4 s, the throughput of RN-TCP is twice more than RR-TCP(DSACK-TA), eleven times more than SACK and eight times more than DSACK-R. RN-TCP outperforms SACK, DSACK-R and RR-TCP(DSACK-TA) irrespective of the propagation delay

## XII. PERFORMANCE: ECN ENABLED MULTIPLE FTP FLOWS

In this section, we conducted a simulation on the same topology as in Figure VI, but used RED queues instead of

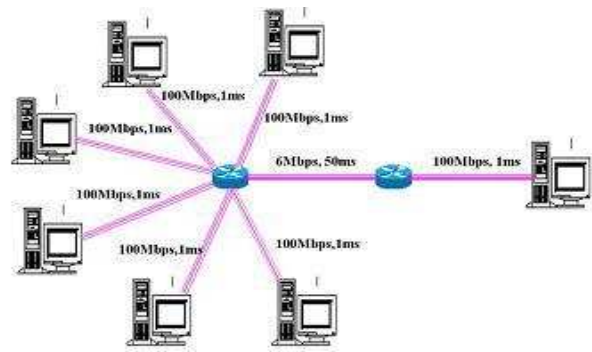


Fig. 22. RN-TCP: Complex Topology

*drop-tail* queues. The gateways were ECN enabled with a queue size set to 200 packets. The process of delaying a packet was normally distributed with a mean packet delay of 25 ms and standard deviation of 8 ms, such that the delay introduced varied from 0 ms to 50 ms. The total number of flows traversing the network were increased to forty flows, in which the sender was configured to send ten SACK flows, ten DSACK-R flows, ten RR-TCP(DSACK-TA) flows and ten RN-TCP flows. All the flows were enabled with ECN. However, when the number of flows are increased, congestion will be caused in the bottleneck queue causing large number of packet drops. The graphs in Figure 21 present the results of the average throughput of SACK, DSACK-R, RR-TCP(DSACK-TA) and RN-TCP flows. It can be seen from the graph, that when there is no reordering the average throughput of all the flows are almost similar. All protocols perform more or less similar to SACK. When the packet delay rate is increased, protocols such as SACK and DSACK-R that does not avoid false retransmits have a reduced throughput performance. Thus protocols that improve the performance when packets get reordered utilize the unused bandwidth available and give a better performance. When the packet delay rate is 1%, the average throughput of RR-TCP(DSACK-TA) is lower compared to RN-TCP, DSACK-R and SACK. This is due to the fact that RR-TCP(DSACK-TA) experiences more timeout events compared to the other protocol due to an increased value of *dupthresh*, whereas RN-TCP fluctuates between three (when packets get dropped) and a higher *dupthresh* value (when packets get reordered). When the packet delay rate is increased, RN-TCP gives a better performance compared to the other protocols. For example, when the packet delay rate is 1%, RN-TCP gives a 20% improvement over RR-TCP(DSACK-TA), 6% improvement over DSACK-R and a 15% improvement over SACK. Similarly when the packet delay rate is 10%, RN-TCP gives a 7% improvement over RR-TCP(DSACK-TA), 27% improvement over DSACK-R and a 44% improvement over SACK.

### A. Performance: Mixture of large FTP flows and small FTP flows

In today's Internet, the majority of the traffic constitutes of file transfers. The average transferred file size is around 10

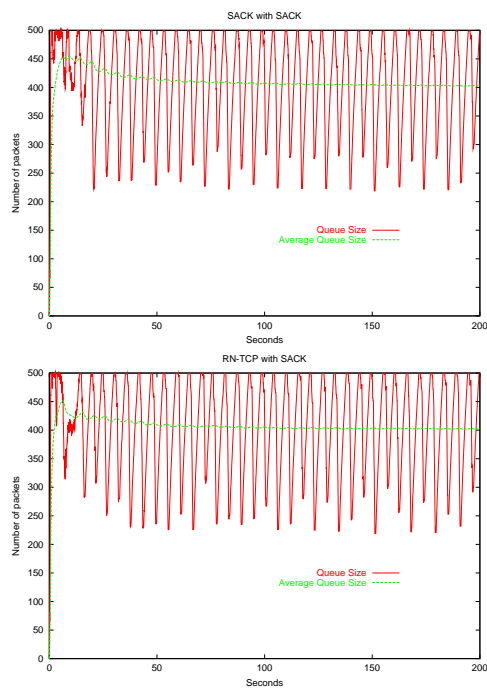


Fig. 23. Queue Size and Average Queue Size.

KB i.e. not more than 10 packets assuming the packet size is 1000 bytes [6]. These type of small files are called *mice*. But the Internet is dominated by large files called *elephants*. To determine whether the performance of RN-TCP is similar to SACK in an environment where there are no reordering events and there are large number of *mice* along with small number of *elephants*, we conducted a simulation on a new network where several sources share a common bottleneck link and a common destination. Figure XII-A shows, six source nodes connected to the bottleneck router with a bandwidth of 100 Mbps and a delay of 1 ms. The bottleneck router was connected to another router with a bandwidth of 6 Mbps and a delay of 50 ms, which in turn was connected to the destination node with a bandwidth of 100 Mbps and a delay of 1 ms. Each source node generated 8 large FTP flows along with 100 small FTP flows. The distribution of inter-arrival times of new connections were taken to be exponential. The file size of the small FTP flows were configured using a Pareto distribution with mean size of 10 KB and shape parameter of 1.5. The packet size was set to 1000 bytes. The experiments were run for 200 seconds. When we configured 8 large SACK FTP flows along with 100 small SACK FTP flows, the average throughput of the large SACK FTP flows were almost 2.67 MB and the average throughput of the small SACK FTP flows were 9.3 KB. When we configured 8 large RN-TCP FTP flows along with 100 small SACK FTP flows, the average throughput of the large RN-TCP FTP flows were almost 2.66 MB and the average throughput of the small SACK FTP flows were 9.3 KB. Moreover, from Figure 23, it can be seen that the average queue size of both experiments are almost similar. If RN-TCP did not reduce the *cwnd* upon a packet drop, then RN-TCP would be aggravating the congestion in the bottleneck link. Thus it is evident that in

the absence of reordering, RN-TCP performs almost similar to SACK.

### XIII. SUMMARY

In this paper, we proposed a proactive solution that prevents the unnecessary retransmits that occur due to reordering events in networks, by allowing the RN-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 in the gateway. We also compared RN-TCP with other protocols namely TCP SACK, DSACK-R and RR-TCP, showing that our solution improves the throughput performance of the network to a large extent.

We believe the gateway could be modified to send the dropped information in an ICMP message to the sender. This requires further study and testing. 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. We believe RN-TCP can be built in a receiver side fashion where the RN-TCP receiver identifies the amount of reordering that has occurred in the network and informs the RN-TCP sender about this information. The RN-TCP sender can then increase the value of *dupthresh* by some value 'k' according to the degree of reordering. Moreover, the simulated results presented in this paper needs verification in the real network.

### REFERENCES

- [1] M. Allman, H. Balakrishnan, S. Floyd, Enhancing TCP's Loss Recovery using Limited Transmit, RFC 3042, January 2001.
- [2] M. Allman, V. Paxson, On Estimating End-to-End Network Path Properties, Proceedings of the SIGCOMM, Cambridge, Massachusetts, USA, pp. 263 - 274, August 1999.
- [3] ATM Forum, ATM User Network Interface (UNI) Signalling Specification Version 4.1, af-sig-0061.002, April 2002, available from [www.atmforum.com/standards/approved.html](http://www.atmforum.com/standards/approved.html).
- [4] J. Bennett, C. Partridge, N. Shectman, Packet Reordering is Not Pathological Network Behaviour. IEEE/ACM Transactions on Networking, Volume 7, Issue 6, pp. 789 - 798, December 1999.
- [5] E. Blanton, M. Allman, On Making TCP More Robust to Packet Reordering, ACM Computer Communication Review, Volume 32, Issue 1, pp. 20 - 30, January 2002.
- [6] M.E. Crovella, M.A.A. Bestavros, Self-similarity in World Wide Web traffic: evidence and possible causes, Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Philadelphia, Pennsylvania, USA, pp. 160 - 169, May 1996.
- [7] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, July 2000.
- [8] V. Jacobson, Congestion Avoidance and Control, Symposium proceedings on Communications architectures and protocols, California, pp. 314 - 329, August 1988.
- [9] D. Katabi, M. Handley, C. Rohrs, Congestion Control for High Bandwidth-Delay Product Networks, Proceedings on ACM SIGCOMM, Pittsburgh, Pennsylvania, USA, pp. 89 - 102, August 2002.
- [10] R. Ludwig, R. Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, Computer Communication Review, Volume 30, Issue 1, pp. 30-36, January 2000.
- [11] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgement Options, RFC 2018, October 1996.

- [12] S. McCanne, S. Floyd, Network Simulator. <http://www.isi.edu/nsnam/ns/>
- [13] J. Mogul, Observing TCP Dynamics in Real Networks, Proceedings of the SIGCOMM, Maryland, USA, pp. 305 - 317, October 1992.
- [14] V. Paxson, End-to-end routing behavior in the Internet, Proceedings of ACM SIGCOMM, Palo Alto, California, USA, pp: 25 - 38, Aug. 1996.
- [15] J. Postel, Transmission Control Protocol, RFC 793, September, 1981.
- [16] J. Postel, Internet Control Message Protocol, RFC 792, September, 1981.
- [17] "Question on XCP", <http://www.postel.org/pipermail/end2end-interest/2005-February/004606.html>.
- [18] K.K. Ramakrishnan, S. Floyd, D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, September 2001.
- [19] "Routers accessing TCP header", <http://www.postel.org/pipermail/end2end-interest/2005-February/004661.html>
- [20] A. Sathiaselan, T. Radzik, RD-TCP: Reorder Detecting TCP. Proceedings of the 6th IEEE International Conference on High Speed Networks and Multimedia Communications HSNMC'03, Portugal, July 2003 (LNCS 2720, pp.471-480).
- [21] A. Sathiaselan, T. Radzik, Improving the Performance of TCP in the Case of Packet Reordering, 7th IEEE International Conference on High Speed Networks and Multimedia Communications HSNMC'04, Toulouse, France July 2004 (LNCS 3079, pp. 63-75).
- [22] 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), pp. 91103, Feb. 1995.
- [23] 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), Atlanta, Georgia, USA, November 2003.