

Reorder Notifying TCP (RN-TCP) with Explicit Packet Drop Notification (EPDN)

Arjuna Sathiaselalan and Tomasz Radzik

Department of Computer Science, King's College London,
Strand, London WC2R 2LS
Tel: +44 20 7848 2841
Email: arjuna,radzik@dcs.kcl.ac.uk
Technical Report: TR-04-08

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. 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.

1 Introduction

Packet reordering occurs naturally as a result of *local parallelism* [2]: a packet can traverse through multiple paths within a device. Packet reordering also occurs mainly due to route changes: if the new route offers a lower delay than the old one, then reordering occurs [10]. A network path that suffers from persistent packet reordering will have severe performance degradation.

TCP has two basic methods of finding out that a segment has been lost.

Retransmission timer

If an acknowledgement 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 [11].

Fast Retransmit

When a TCP sender receives three *dupacks* (duplicate acknowledgements) 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 [6]. Fast Retransmit uses a parameter called *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 [3]:

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 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).

Section 1.1 presents the previous work related to our study. Section 2 presents the details of our proposed solution. In Sections 6,7, 8 , 9, 10 and 11 , 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 12.

1.1 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 [8]. The Eifel algorithm is robust to up to a *cwnds* 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.[8] 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 [5]. 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 con-

taining 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.

- In [3], 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 [13], 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.

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 1. RR-TCP Algorithms

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

- In [12], we proposed a novel method 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 [13] and the method we presented in [12] 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.

2 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 TCP receiver about the dropped information as Explicit Packet Drop Notification Version 2.0 (EPDNv2.0).

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 current 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 reordering or not and informs the RN-TCP

sender about its assumption. (Informing the sender is done by setting the *drop-negative* bit in corresponding DUPACKs.) If the packets had been dropped in the network, the TCP sender retransmits the lost packets after waiting for three DUPACKs. If the packets are assumed to be reordered in the network, the sender 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 4. Assume a data packet $Q = \langle 1, 2, \dots, data_Q \rangle$ is last received packet in the receiver buffer queue. When the receiver receives the next data packet $P = \langle 1, 7, 5, 4, data_P \rangle$, 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 the $P:min = 4$ and the last received packet $Q:n = 2$ in the receiver's buffer queue or a gap in between $P:max = 5$ and the current received packet $P:n = 7$, then the packets within the gap (3,6) have been probably reordered in the network and packets (4,5) 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, this means that the packets in between the last received packet $Q:n$ and the current packet $P:n$ have been probably reordered.

3 EPDNv2.0: Details of the implementation

3.1 Data Structure used.

Each gateway has a hashtable storing entries $E = \langle flowid, max, min \rangle$, where *flowid* is the flow id (also the key used to index the hashtable), *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.

3.2 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 packets of this flow have been already dropped), then this entry is updated as follows:

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

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

- 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

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

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

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

3.3 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,
$$P:max = \max\{E:max, P:max\}, \text{ and } P:min = \min\{E:min, P:min\}.$$

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.

3.4 EPDN: Implementation Issues

- 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. Every EPDN enabled router, decrements the EPDN-TTL along with the actual TTL (Time To Live) 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 [7] and is based on discussions from [14].
- According to Chinoy [4], most routing changes occur at the edges of the network and not along its 'backbone'. This ensures that the routing of packets

do not change inside the backbone more often, allowing the the maximum-minimum dropped information to be inserted into the next packet that passes through the gateway. In case 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 in an ICMP message to the sender. This requires further study and testing.

4 RN-TCP: Details of the Implementation

4.1 Receiver side: Implementation details

The TCP receiver maintains two lists: the reorder list and the drop list. The elements of these lists are packet sequence numbers. The data packets that arrive at the receiver could bring in the maximum-minimum dropped information about any dropped packets irrespective of the sequence. For example, a packet with lesser sequence number could bring in higher minimum and maximum dropped sequence numbers for that particular flow. The TCP receiver has to consider all possible cases before considering whether the gaps caused are due to reordering or dropped packets.

TCP receiver algorithm: Processing data packets. When a data packet P arrives at the TCP receiver, the following computation is done. The 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 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 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 TCP receiver assumes that those packets have been dropped.
- * If not, then the packets within the gap are assumed to be reordered. The TCP receiver adds those sequence numbers required to fill the gap to the reordered list.
- If $P:n$ is less than $Q:n$, then the TCP receiver checks if $P:n$ is in the reordered list. If yes, then the packet is removed from the reordered list.
- If $P:min$ and $P:max$ are not null, the TCP receiver checks if $P:n$ is in the reordered list. If present, then the entry is deleted. Then the TCP receiver checks if $P:n > Q:n$.
 - If $P:n > Q:n$,
 - * The TCP receiver checks if $P:min > Q:n$ AND $P:max < P:n$.
The 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 TCP receiver adds those sequence numbers required to fill the gap to the reordered list. Whilst adding, check if the sequence numbers from $P:min$ to $P:max$ are present in the reordered list. If present, remove them from the reordered list. Add $P:min$ to $P:max$ into the droplist.
 - * If $P:min > Q:n$ AND $P:max > P:n$ and $P:min < P:n$.
The TCP receiver checks for a gap between $P:min$ and $Q:n$. If there is a gap, the TCP receiver adds those sequence numbers required to fill the gap to the reordered list. Then the TCP receiver checks if the sequence numbers from $P:min$ to $P:n$ and $P:max$ are present in the reordered list. If present, remove them from the reordered 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.
 - * If $P:min > Q:n$ AND $P:max > P:n$ and $P:min > P:n$.

Add sequence numbers from $P:min$ to $P:max$ into the droplist. Add sequence numbers from $Q:n + 1$ to $P:n - 1$ into the reordered list.

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

The TCP receiver checks for a gap between $P:max$ and $P:n$. If there is a gap, the TCP receiver adds those sequence numbers required to fill the gap to the reordered list. Then the TCP receiver checks if the sequence numbers from $P:min$ to $P:max$ are present in the reordered list. If present, remove them from the reordered list. Add sequence numbers from $P:min$ to $Q:n - 1$ and from $Q:n + 1$ to $P:max$ into the droplist.

- * 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 droplist. Add sequence numbers from $Q:n + 1$ to $P:n - 1$ into the reordered list.

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

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

- If $P:n < Q:n$,

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

The TCP receiver checks for a gap between $P:min$ and $Q:n$. If there is a gap, the TCP receiver adds those sequence numbers required to fill the gap to the reordered list. Put the sequence numbers from $P:min$ to $P:max$ into the drop list for future references.

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

The TCP receiver checks if the sequence numbers from $P:min$ to $Q:n$ are present in the reordered list. If present, remove them from the

reordered 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.

* If $P:min < Q:n$ AND $P:max > P:n$ and $P:min > P:n$.

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

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

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

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, then check if packet $P:n + 1$ is in the reordered list. If yes, the *drop-negative* bit is set and the cumulative ACK is sent. Else, 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 reordered list. If yes, the *drop-negative* bit is set for that particular SACK packet. Else, the SACK is sent without setting the *drop-negative* bit.

4.2 Sender side: Implementation Details

Limited Transmit. The limited transmit algorithm [?] 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 *cwnd*. When a greater *dupthresh* value is used, the sender is allowed to send more packets (*dupthresh* - 1 packets) beyond its current *cwnd*. We extend the limited transmit to send upto 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.

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. 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. The *dupthresh* value is calculated as follows:

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

where *swnd* is the sending window (minimum of *cwnd* and the receiver window). 'k' is a constant which is set based on the RTT as follows:

- If the sampled RTT is in between 0 ms and 50 ms, we set the value of 'k' to be 0.05 and for every false fast retransmit we increase 'k' by 0.05.
- If the sampled RTT is in between 51 ms and 100 ms, we set the value of 'k' to be 0.1 and for every false fast retransmit we increase 'k' by 0.1.
- If the sampled RTT is in between 101 ms and 150 ms, we set the value of 'k' to be 0.15 and for every false fast retransmit we increase 'k' by 0.15.
- If the sampled RTT is in between 151 ms and 200 ms, we set the value of 'k' to be 0.2 and for every false fast retransmit we increase 'k' by 0.2.
- If the sampled RTT is in between 201 ms and 250 ms, we set the value of 'k' to be 0.25 and for every false fast retransmit we increase 'k' by 0.25.

- If the sampled RTT is greater than 250 ms, we set the value of 'k' to be 0.3 and for every false fast retransmit we increase 'k' by 0.3.

These values were determined to be optimal after heuristically testing for different values of 'k'.

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 e.g. multipath routing), and sets the *drop-negative* bits in corresponding DUPACKs. 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 *drop-negative* bit is set) after receiving three DUPACKs (*dupthresh* value is immediately set to three). Moreover the value of 'k' is set back to those initial values based on the sample RTT (discussed in the previous section).

Sender side algorithm: Processing the ACK packets. When an acknowledgement is received, the TCP sender does the following,

- If none of the three DUPACKs received have their *drop-negative* bit set, then the TCP sender assumes that the packet has been dropped. So the sender retransmits the lost packet after receiving three DUPACKs and enters fast recovery.
- If the fourth ACK packet that causes the third DUPACK has a *drop-negative* bit set, then the TCP sender assumes that the packet has been reordered and waits for 'k' more DUPACKs before retransmitting the packet. While

waiting, if the following ACK packet has no *drop-negative* bit then the TCP sender assumes the packet could have been dropped and retransmits the packet immediately and sets the *dupthresh* to 3.

- If the current *dupthresh* value is not enough to prevent a false fast retransmit, increase the *dupthresh* value by 1.
- If the timer runs out while waiting for '3+k' DUPACKs (assuming the value of *dupthresh* is high), then the sender assumes that the packet has been dropped, retransmits the packet and enters fast recovery. The sender also assumes that all contiguous packets following the dropped packet in that particular SACK block are dropped and retransmits the contiguous packets after receiving three DUPACKs.

5 Storage and Computational Costs

The IP options 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 one 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 each packet sequence number, and another 4 bytes for each pointer. This means that the total memory required would be about 320 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 500KB 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 amount of packets the TCP receiver assumes that has 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 costs $O(m)$ 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. If the list is empty, then the computational cost is constant time.

We 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.



Fig. 1. RN-TCP:Simulation Scenario

6 Topology

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 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 queueing strategy 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.

7 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 6Mbps capacity with a propagation delay of 50 ms. The process of delaying a packet was normally distributed with a mean 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.

7.1 Throughput: Varying Packet Delay Rate

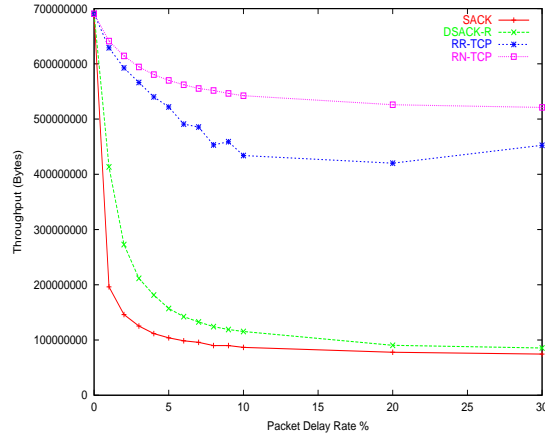


Fig. 2. Throughput versus fraction of delayed packets.

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 and RN-TCP.

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 for all packet delay rates. For e.g., when the link experiences 5% of packet delays, RN-TCP's throughput performance is almost 9% more than RR-TCP, 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, 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

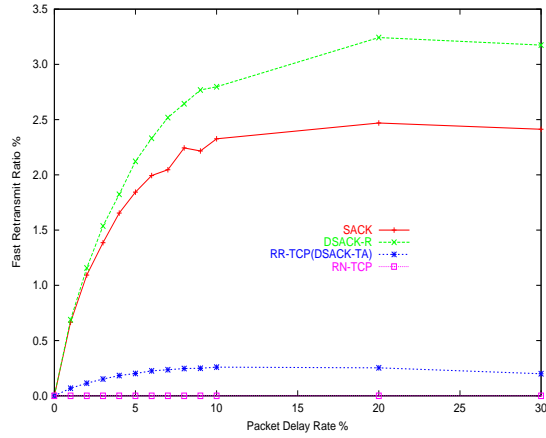


Fig. 3. Fast retransmit ratio versus fraction of delayed packets.

followed by RR-TCP 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.

7.2 Steady State Congestion Window

The graphs in Figure 4 present the comparison of the *cwnd* states of SACK versus RN-TCP, DSACK-R versus RN-TCP and RR-TCP versus RN-TCP when 30% of packets were delayed individually for each of these protocols. When packets get delayed frequently in the network, there are unnecessary false fast retransmits causing the *cwnd* of SACK to reduce very often and thus the *cwnd* is not able to reach the maximum window size of 50 segments. DSACK-R has the ability to retrieve to the previous *cwnd* on detecting a false retransmission. This is not suf-

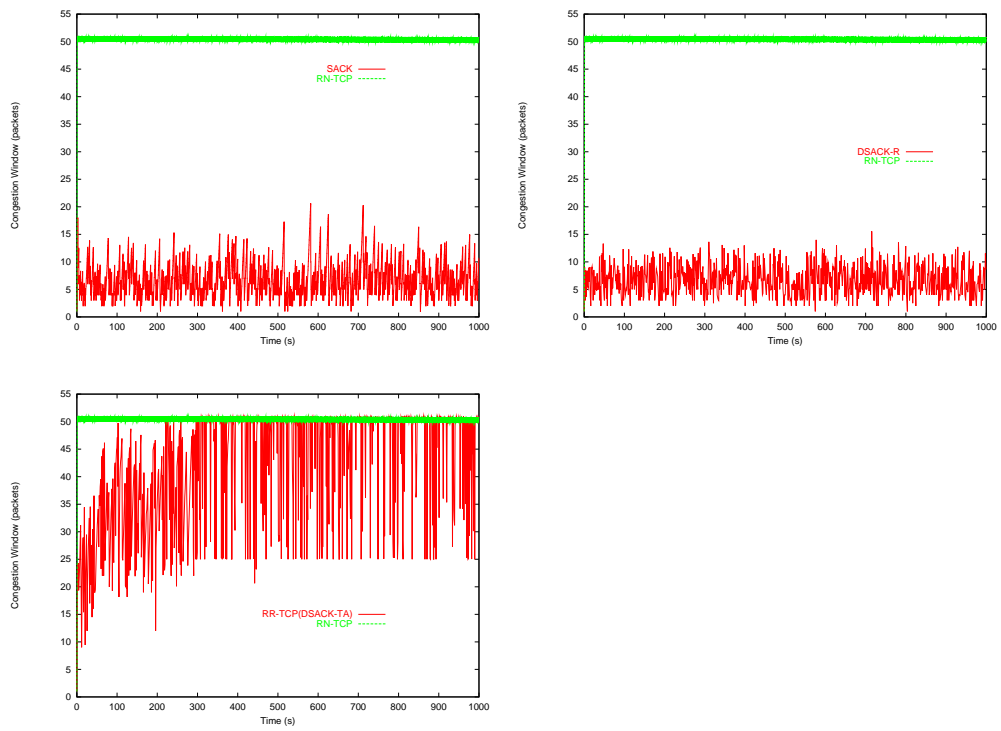


Fig. 4. Comparison of congestion window.

efficient to reach the maximum *cwnd* since DSACK-R does not proactively detect packet reorders and the *cwnd* is constantly reduced upon a false retransmission. RR-TCP is able to achieve the maximum *cwnd* size only after 300 seconds. It is evident from the Figure 4, RR-TCP does not maintain a steady *cwnd* state and keeps on fluctuating whereas RN-TCP is able to distinguish a reorder event and prevent unnecessary reduction of the *cwnd* when packet reordering occurs. RN-TCP maintains a steady *cwnd* throughout the experiment.

7.3 Link Utilization

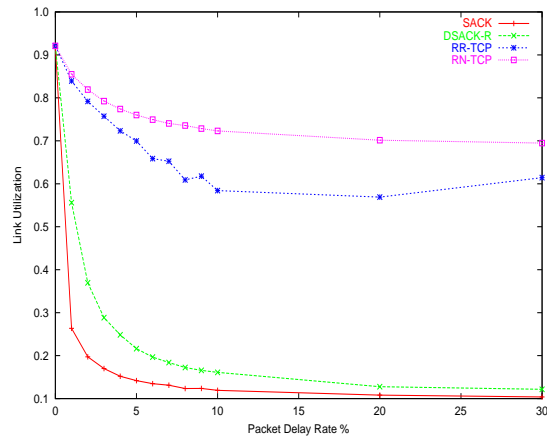


Fig. 5. Link Utilization versus fraction of delayed packets.

Figure 5, presents the link utilization of SACK, DSACK-R, RR-TCP and RN-TCP with varying packet delay rate. It is evident from the figure, that RN-TCP offers better link utilization when compared to SACK, DSACK-R and RR-TCP.

The graphs in Figure 6, present the comparison of the link utilization of SACK versus RN-TCP, DSACK-R versus RN-TCP and RR-TCP versus RN-

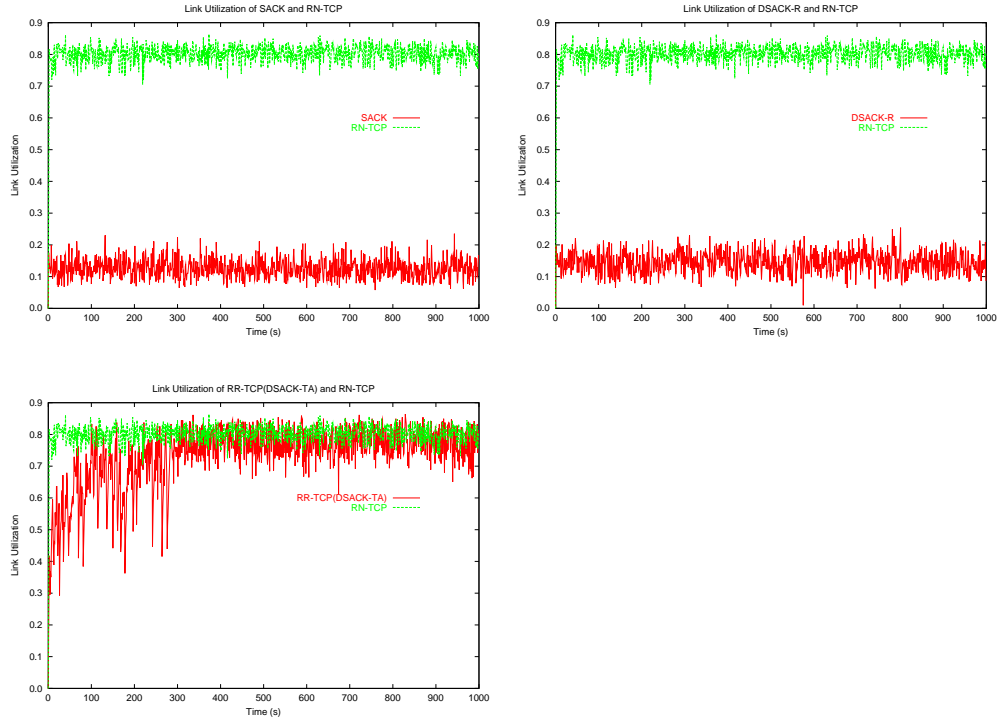


Fig. 6. Link utilization.

TCP when 30% of packets were delayed. When packets get delayed in the network, there are unnecessary false fast retransmits causing the *cwnd* of SACK and DSACK-R to reduce by half. This causes the sender to send lesser amount of packets resulting in improper utilization of the available bandwidth. RR-TCP is able to achieve 80% link utilization only after 300 seconds. Moreover RR-TCP does not maintain a steady utilization rate throughout the experiment. RN-TCP prevents unnecessary reduction of the *cwnd* and is able to maintain a steady utilization rate of almost 80% throughout the experiment.

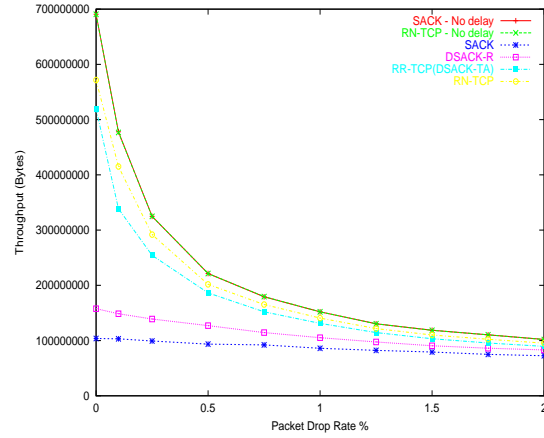


Fig. 7. Throughput versus fraction of dropped packets. 5% of packets delayed.

7.4 Throughput: Varying Packet Drop Rate

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP and RN-TCP when the link experiences both packet drops and packet delays. We also compared the performance of SACK and RN-TCP with packet drops only. 5% of the packets were delayed. The packet drop rate varied from 0% to 2%. Figure 7, reveals that the throughput of SACK, DSACK-R, RR-TCP and RN-TCP reduces 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 reduce 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, 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.

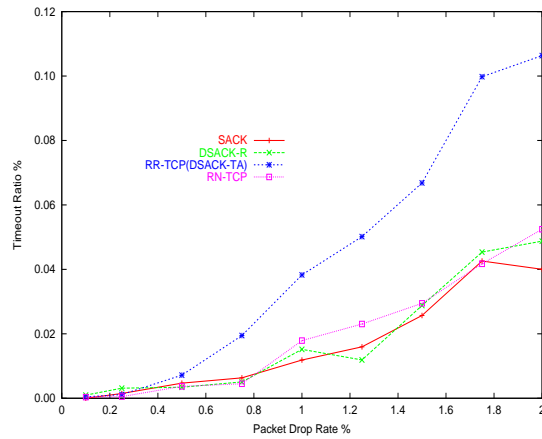


Fig. 8. Timeout ratio versus fraction of dropped packets.

It is also evident from figure 8, that the number of timeout events of RN-TCP is much less compared to RR-TCP. RR-TCP does not have the ability to detect whether a packet has been dropped or reordered instantaneously, leading to more timeouts when the *dupthresh* value is high. RN-TCP's ability to distinguish packet loss from packet reordering reduces unnecessary timeouts by setting the *dupthresh* value to three upon a packet drop. In case a packet drop is detected after a timeout event, RN-TCP 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 'drop-negative' bit is set) after receiving three DUPACKs (*dupthresh* value is immediately set to three).

7.5 Fairness

When a protocol gives an throughput performance improvement over a standard protocol, the question of fairness arises. According to [13], the poor throughput

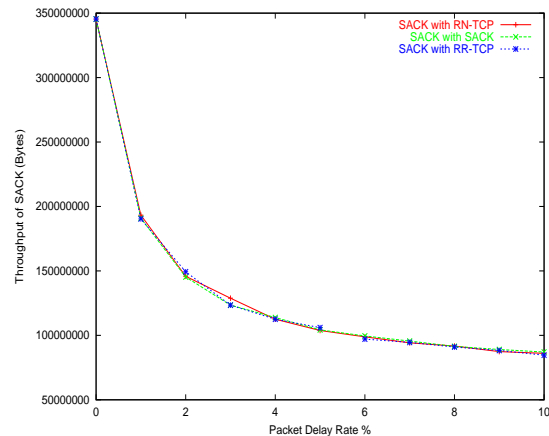


Fig. 9. Throughput of SACK while co-operating with RN-TCP and RR-TCP(DSACK-TA) versus fraction of reordered packets

performance of SACK is due to the reordering of packets that occur on the network. The enhanced versions such as DSACK-R, RR-TCP 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 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 of 50 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 that packets do not get dropped by having a suitable queue size. From the figure 9, 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 and SACK with RN-TCP perform almost similar to SACK with SACK.

7.6 Throughput: ACK Reordering

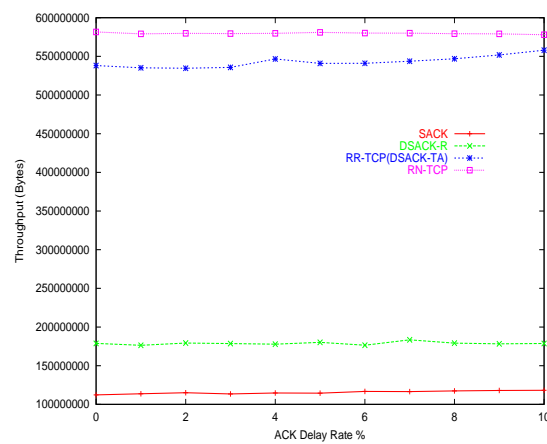


Fig. 10. Throughput versus fraction of reordered ACK packets, 4% of data packets delayed.

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 and RN-TCP when the link experiences both data packet delays and ACK packet delays. We delayed 3% of data packets and varied the delay of ACK packets from 1% to 10%. From the Figure 10, 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 and RN-TCP.

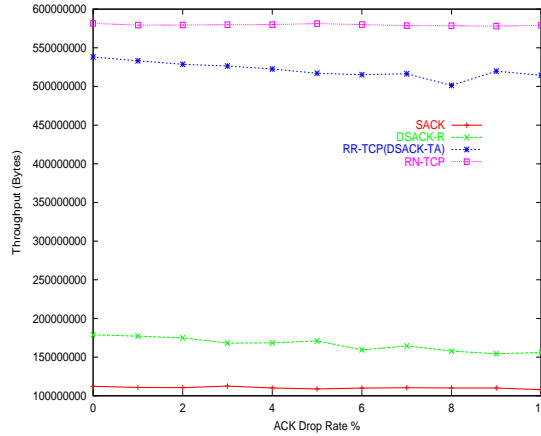


Fig. 11. Throughput versus fraction of dropped ACK packets, 4% of data packets delayed.

7.7 Throughput: ACK Drops

We would also like to verify whether loss of ACK packets has any effect on the throughput performance of the sender. In this section, we examine the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP 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 11, it is evident that when ACK packets get dropped, there is no significant impact on the throughput of all the protocols.

7.8 Throughput: Various Reordering Distribution

The graphs in figure 12, present the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP 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 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 independent of any reordering distribution.

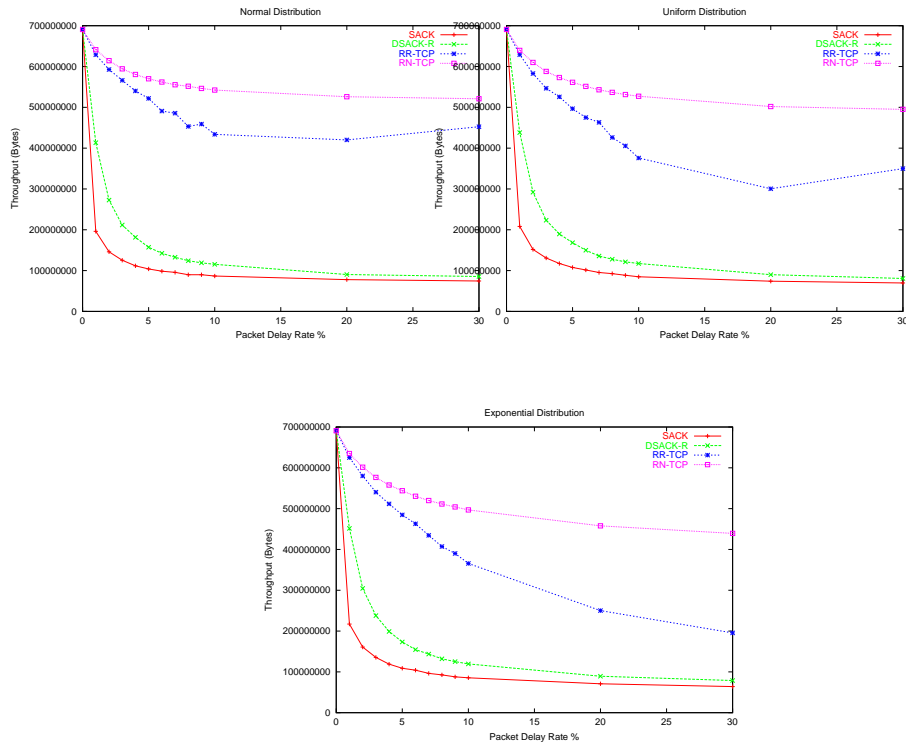


Fig. 12. Various probability distributions.

8 Multipath Routing

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP and RN-TCP when multipath reordering occurs. The link between the routers was set to 6Mbps capacity with a delay of 50 ms. We varied the average delay from 0.0 seconds to 0.2 seconds (upto $2 \times RTT$).

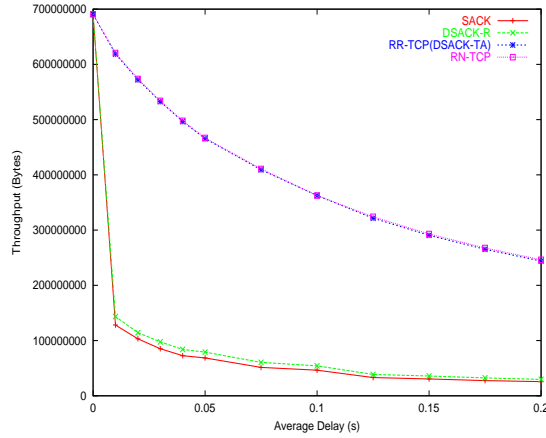


Fig. 13. Throughput versus average delay.

The average delay represents the RTT difference between the 100 ms RTT path and the longer path. When the average 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 delay is increased, the throughput of SACK and DSACK-R reduce considerably, whereas the throughput of RR-TCP and RN-TCP reduce much more slowly. RN-TCP outperforms SACK and DSACK-R and achieves similar throughput to RR-TCP.

9 Severe Reordering

In this section, we compare the throughput performance of the simulated network using SACK, DSACK-R, RR-TCP and RN-TCP when packets experience large delay distributions in the order of multiples of RTT. The link between the routers was set to 1.3Mbps 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 of yP ms (P is the propagation delay) and standard deviation of $\frac{y}{3}P$

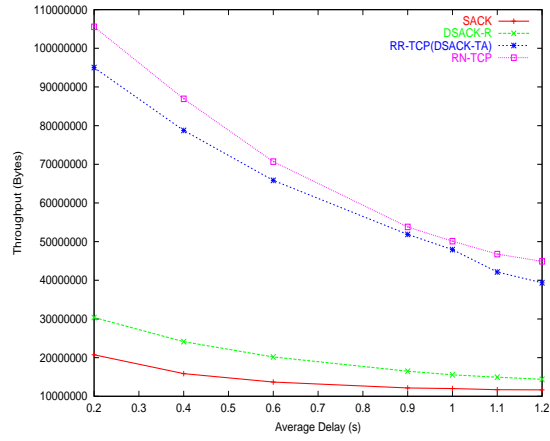


Fig. 14. Throughput versus average delay.

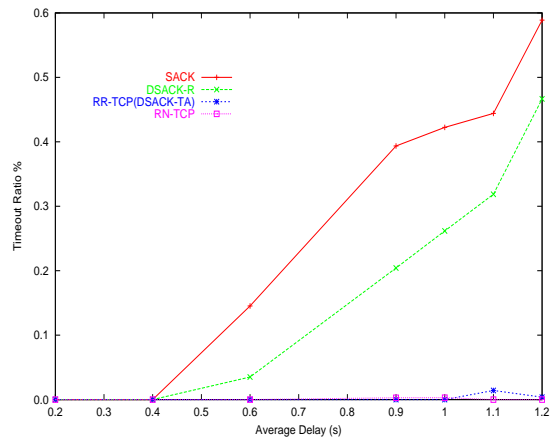


Fig. 15. Timeout ratio versus average delay.

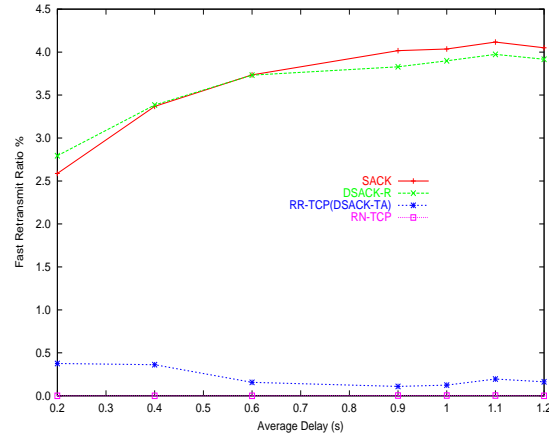


Fig. 16. Fast retransmit ratio versus average delay.

ms, such that the delay introduced varied from 0 and $2yP$ seconds. The packet delay rate was fixed at 7%. We varied the value of y from 1.0 to 6.0. From the Figure 14, when packets experience an average delay of 0.2 s, the throughput of RN-TCP is 14% more than the throughput of RR-TCP, almost five times more than SACK and four times more than DSACK-R. When packets experience an average of 1.2 s, the throughput of RN-TCP is 18% more than RR-TCP, four times more than SACK and three times more than DSACK-R.

Moreover, RN-TCP does not experience any timeout events whereas RR-TCP experiences timeouts when the average delay is more than 1.0 s. Both SACK and DSACK-R experiences more timeouts when the average delay is increased. RN-TCP does not undergo any fast retransmission when compared to RR-TCP. SACK and DSACK-R undergo large number of congestion window reductions due to fast retransmits.

10 Varying bandwidth with constant delay

In this section, we compare the throughput performance of SACK, DSACK-R, RR-TCP and RN-TCP with packet reordering events for various link capacities varying from $[1.3Mbps, 10Mbps]$. The delay was fixed at 80 ms. 10% of packets were delayed using uniform distribution $[0, 2P]$ where P is the propagation delay.

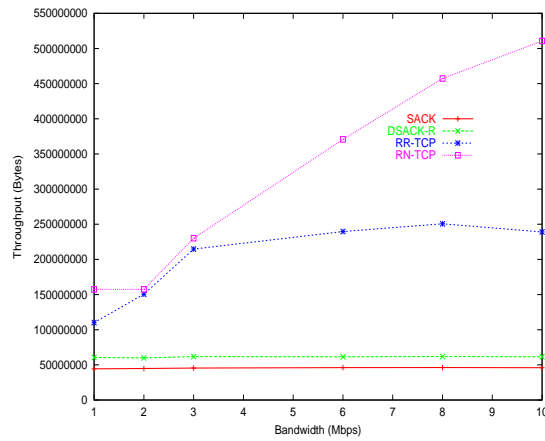


Fig. 17. Throughput versus varying bandwidth, 80 ms propagation delay.

As shown in the Figure 17, when the link capacity is set to 3Mbps, the throughput of RN-TCP is almost 7% more than RR-TCP, five times more than SACK and three times more than DSACK-R. When the link capacity is set to 10Mbps, the throughput of RN-TCP is twice more than RR-TCP, 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 irrespective of the link capacity.

11 Varying delay with constant bandwidth

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

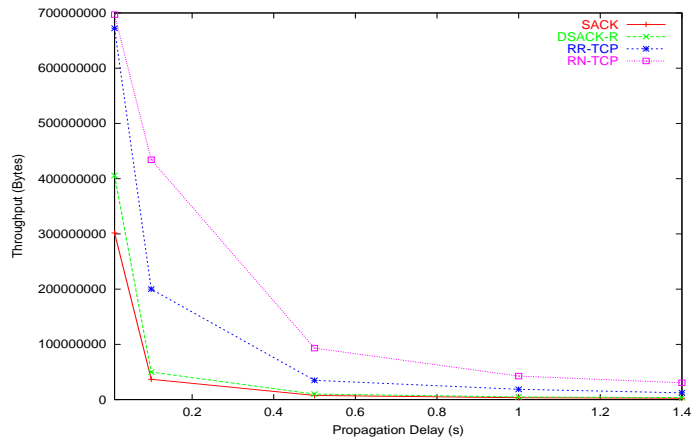


Fig. 18. Throughput versus varying delay, 10Mbps bandwidth.

As shown in the Figure 18, when the link delay is set to 10 ms, the throughput of RN-TCP is almost 4% more than RR-TCP, 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, eleven times more than SACK and eight times more than DSACK-R. RN-TCP outperforms SACK, DSACK-R and RR-TCP irrespective of the propagation delay.

12 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

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 TCP receiver identifies the amount of reordering that has occurred in the network and informs the TCP sender about this information. The 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. Allman, M., Paxson, V.: On Estimating End-to-End Network Path Properties. Proceedings of the SIGCOMM (1999)
2. Bennett, J., Partridge, C., Shectman, N.: Packet Reordering is Not Pathological Network Behaviour. IEEE/ACM Transactions on Networking (1999)
3. Blanton, E., Allman, M.: On Making TCP More Robust to Packet Reordering. Proceedings of the SIGCOMM (2002)
4. Chinoy, B.: Dynamics of Internet Routing Information. Proceedings of the SIGCOMM (1993)

5. Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M.: An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (2000)
6. Jacobson, V.: Congestion Avoidance and Control. Proceedings of the SIGCOMM (1988)
7. D. Katabi, M. Handley, C. Rohrs, Congestion Control for High Bandwidth-Delay Product Networks,. Proceedings on ACM SIGCOMM 2002.
8. Ludwig, R., Katz, R.: The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. Computer Communication Review, 30(1)(2000)
9. McCanne, S., Floyd, S.: Network Simulator. <http://www.isi.edu/nsnam/ns/>
10. Mogul, J.: Observing TCP Dynamics in Real Networks. Proceedings of the SIGCOMM (1992).
11. Postel, J.: Transmission Control Protocol. RFC 793 (1981)
12. Sathiaselalan, A., Radzik, T.: 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).
13. Zhang, M., Karp, B., Floyd, S., Peterson, L.: RR-TCP: A Reordering-Robust TCP with DSACK. IEEE International Conference on Network Protocols (2003).
14. Question on XCP, <http://www.postel.org/pipermail/end2end-interest/2005-February/004606.html>