Improving the performance of TCP in the case of packet reordering

Arjuna Sathiaseelan and Tomasz Radzik

Department of Computer Science, King's College London,
Strand, London WC2R 2LS
{arjuna,radzik}@dcs.kcl.ac.uk

Abstract. Numerous studies have shown that packet reordering is common, especially in high speed 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 an effective 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.

1 Introduction

Research on the implications of packet reordering on TCP networks indicate that packet reordering is not a pathological network behavior. For example, packet reordering occurs naturally as a result of *local parallelism* [8]: 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. Packet reordering occurs mainly due to route changes: if the new route offers a lower delay than the old one, then reordering occurs [15]. A network path that suffers from persistent packet reordering will have severe performance degradation.

TCP receivers generate cumulative acknowledgements which indicate the arrival of the last in-order data segment [9]. For example, assume that four segments A, B, C and D are transmitted through the network from a sender to a receiver. When segments A and B reach the receiver, it transmits back to the sender an *ack* (acknowledgement) for B which summarizes that both segments A and B have

been received. Suppose segments C and D have been reordered in the network. At the time segment D arrives at the receiver, it sends the *ack* for the last in-order segment received which in our case is B. Only when segment C arrives, the *ack* for the last in-order segment (segment D) is transmitted.

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 [9].

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 [4]. 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 [10]:

1. When a network path reorders data segments, it may cause the TCP receiver to send more than three successive *dupacks*, and this triggers the Fast Retransmit procedure at the TCP sender for data segments that may not necessarily be lost. Unnecessary retransmission of data segments means that some of the bandwidth is wasted.
2. The TCP transport protocol assumes congestion in the network 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. If multiple retransmits occur for a single window, the congestion window decreases quickly to a very low value, and the rate of transmission drops significantly.

3. TCP ensures that the receiving application receives data in order. Persistent reordering of data segments is a serious burden on the TCP receiver since the receiver must buffer the out-of-order data until the missing data arrive to fill the gaps. Thus the data being buffered is withheld from the receiving application. This causes unnecessary load to the receiver and reduces the overall efficiency of the system.

We used a network simulator to get an indication about the extent of the deterioration of the TCP performance when reordering of packets occurs. We tested our network to determine the average throughput as a function of random queue swaps performed every 1 or 8 seconds. For e.g. when there was no reordering in the network, the average throughput of a 10 minute TCP connection was 607500 bytes/second. When reordering was introduced say 15 queue swaps every second, the average throughput was 526000 bytes/second. Thus the throughput reduced to about 13.4%. When we increased the amount of swaps every 1 or 8 seconds, the average throughput gradually decreased.

We propose extending the TCP protocol to enable TCP senders to recognize whether a received *dupack* means that a packet has been dropped or reordered. The extended protocol is based on storing at the gateways information about dropped packets and informing the receiver about dropped packets. We term this mechanism of informing the receiver about dropped packets as Explicit Packet Drop Notification (EPDN). Based on this information the sender is notified whether the packet has been dropped or reordered. We call this protocol RD-TCP (Reorder Detecting TCP).

Section 2 presents the previous work related to our study. Section 3 presents the details of our proposed solution. In Sections 4, 5 and 6, 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 7.

2 Related Work

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

– The Eifel algorithm uses the TCP time stamp option to distinguish an original transmission from an unnecessary retransmission [5].
– A method has been proposed by [7], for timing the *ack* of a segment that has been retransmitted. If the *ack* returns in less that $3/4 \times RTTmin$, the retransmission is likely to be spurious.
– The DSACK option in TCP, allows the TCP receiver to report to the sender when duplicate segments arrive at the receiver's end. Using this information, the sender can determine when a retransmission is spurious [6]. In the DSACK proposal, the current congestion window is stored before reducing the congestion window upon detection of a packet loss. Upon an arrival of a DSACK, the TCP sender can find out whether the retransmission was spurious or not. If the retransmission was spurious, then the slow start threshold is set to the previous congestion window. We compared the performance of DSACK with RD-TCP and have presented the results in the results section.
– [10] use the DSACK information to detect whether the retransmission is spurious and propose various techniques to increase the value of *dupthresh* when the sender detects a spurious retransmit. 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 our proposal, we enable the TCP sender to distinguish whether a packet has been dropped or reordered and we increase the *dupthresh* to '3+k' *dupacks* or decrease to three *dupacks* accordingly.
– In [16], we had proposed a novel method to enable the TCP senders to distinguish whether a packet has been dropped or reordered in the network. If the packet had been reordered in the network, we increase the value of dupthresh by waiting for '3+k' *dupacks* before retransmitting the packet. Our method was proposed for networks that follow symmetric routing and did not consider the case of asymmetric routing. The current paper can be viewed as simplification and generalization of our work presented in [16].

These methods, with exception of the method we presented in [16] 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.

3 Our proposed Solution

When the TCP sender sends data segments to the TCP receiver through intermediate gateways, these gateways drop the incoming data packets when their queues are full or reach a threshold value. Thus the TCP sender detects congestion only after a packet gets dropped in the intermediate gateway. When a packet gets reordered in the gateway or path, the TCP sender finds it impossible to distinguish whether the data packet has been dropped or reordered in the network. In this paper, we try to solve this problem by proposing a solution to distinguish whether the packet has been lost or reordered, by having a hashtable that maintains for each flow the maximum sequence number and minimum sequence number of the packets that gets dropped in the gateway. 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 (EPDN). (The following case is just a generalization of one particular case where the current received packet at the TCP receiver is greater than the last received packet in the buffer queue. We have provided the algorithm in Section 3.1 for various generalizations.)When the TCP receiver receives a data packet, the maximum-minimum entries are checked. A gap in between the minimum sequence number and the last received packet in the receiver's buffer queue or a gap in between the maximum dropped entry and the current received packet means that the packets within those gaps

have been probably reordered in the network and not dropped. If there are no gaps, it means that most likely all the packets between the last received packet and the recently received packet have been dropped at the gateway. Alternatively, if the TCP receiver receives a data packet, and if the maximum-minimum entries are empty, this ensures that the packets in between the last received packet and the current packet have been probably reordered. Thus the TCP receiver can determine whether the gaps between the out of order packets are caused by reordering or not and can inform the TCP sender about the cause. If the packets had been dropped in the network, the TCP sender retransmits the lost packets after waiting for 3 *dupacks*. If the packets had been reordered in the network, the TCP sender waits for '3+k' *dupacks* before retransmitting the packets. We term our new version of TCP as RD-TCP (Reorder Detecting TCP).

Assume a network with source node A and destination node B with intermediate gateways R1 and R2. Node A sends data packets P_1, P_2, P_3, P_4, P_5 to node B through the gateways R1 and R2. If R1 drops packet P_2, P_3 due to congestion in the network, the minimum packet sequence number entry will be P_2 and the maximum sequence number entry will be P_3. When P_4 passes through the gateway, the maximum and minimum entries are inserted into the maximum and minimum dropped entries of P_4 respectively and the entries in the hashtable are deleted. On receipt of packet P_4, node B checks whether the packet is an out of order packet. As the packet is out of order, the TCP receiver checks for the maximum and minimum sequence numbers. As there are no gaps between the minimum packet sequence number entry (P_2) and the last in-order packet P_1, the TCP receiver knows that the packets P_2 and P_3 have been dropped. Thus it does not set any *reordered* bit in the corresponding *dupacks*. When the TCP sender receives these *dupacks* without any *reordered* bit set, the TCP sender A knows that that packet has been dropped and retransmits after 3 *dupacks*.

Suppose the packet P_2 had been reordered and P_3 had been dropped at gateway R1. In the hashtable, the minimum dropped entry would by P_3 and the maximum dropped entry would be P_3. When P_4 passes through the gateway, the maximum and minimum entries are inserted and the entry in the hash table is deleted. When node B receives P_4, the maximum and minimum entries are checked

for gaps. As there is a gap between the last received packet P_1 and the minimum dropped entry of packet P_4 which is packet P_3, the TCP receiver knows that the packet P_2 has been probably reordered in the network. The TCP receiver maintains a list of all reordered packets and the sequence number of P_2 is inserted into the list. Also the TCP receiver sets the *reordered* bit for the corresponding *dupacks*. When the TCP sender receives these *dupacks* with the *reordered* bit set, the TCP sender knows that the packet has been reordered in the network and waits for '3+k' *dupacks* to retransmit the packet. Thus the TCP sender delays the retransmission procedure assuming that the reordered packet would arrive at the TCP receiver. When the TCP receiver receives the reordered packet, the entry in the list is deleted and a cumulative *ack* is sent to the TCP sender.

If a following gateway drops the data packet carrying the dropped information, then the maximum dropped entry for that particular flow in that gateway would be the sequence number of the dropped packet. These values are inserted into the next data packet that passes through the gateway successfully. Thus we ensure that the dropped information is successfully propagated to the TCP receiver.

There could be a possibility of packets being dropped in the gateways while the receiver assumes the packets have been reordered and sets the reordered bits in corresponding *dupacks*. Suppose packets P_1, P_2, P_3 are sent through one path and packets P_4, P_5 and P_6 are sent through an alternate path. If P_2 and P_3 are dropped in the network and if there was no other packet following P_3, then on the arrival of P_4, the receiver checks for the max-min entries and finds it to be empty. Thus the receiver assumes that the packets P_2 and P_3 have been reordered in the network. The receiver sets the reordered bits for the corresponding *dupacks* and the sender waits for '3+k' *dupacks*. If the packets have been dropped and if the sender waits for a longer time i.e. if the value of 'k' is large, then the timer times out and the packet P_2 gets retransmitted and the cwnd is reduced. The TCP sender then assumes that all packets following P_2 till P_4 (the TCP sender can determine the packet gaps using the sack information) have been dropped and retransmits those packets after receiving 3 *dupacks*.

If the value of 'k' is not large enough, then the TCP will continue to send unnecessary retransmissions. If the value of 'k' is set too

large, fast retransmit may not be triggered leading to retransmission timeout. The best value of 'k' depends on the impact of reordering and could be varied depending on the current network conditions as proposed in [10] i.e. if the TCP sender detects spurious retransmits even though it has incremented the value of 'k', then the sender can further increase the value of 'k' to reduce the number of unnecessary retransmissions that occur due to reordering. The TCP receiver can detect the amount of reordering present in the network and can inform the sender about this amount, which in turn can set the value of 'k' based on that value. This requires further research and we are currently working on it. Currently, we have tested our protocol for various values of 'k' and have presented the results for $k = 3$ ($dupthresh = 6$). RD-TCP effectively detects reordering of packets when packets travel through multiple paths or if they get reordered in the gateway. According to Chinoy [2], 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. Thus the maximum-minimum dropped information would always be inserted into the next packet that passes through the gateway.

3.1 Details of the implementation

Data Structure used We use a hashtable to maintain the flow ids and the maximum and minimum dropped packet numbers (max:PNO) and (min:PNO) for the respective flow ids (F_{id}). The flow id is the index and the packet numbers are the items in the list for a particular flow id.

Recording information about dropped packets

- Initially, the hashtable is empty.
- When a packet $< F_{id}, PNO_i >$ gets dropped in the gateway, the corresponding flow id (F_{id}) is used as the index to check the hashtable to find out whether there is an entry for that particular flow.
- If an entry is present (means packets have been already dropped), then sequence number of the dropped packet is inserted as follows:

max:PNO = PNO_i
min:PNO = min:PNO

- If an entry is not present (means this is the first packet to be dropped), an entry is created, and the sequence number of the dropped packet is inserted as follows:
  max:PNO = PNO_i
  min:PNO = PNO_i

Processing the data packets at the gateway When a data packet $< F_{id}, PNO_i >$ arrives and not dropped at the gateway,

- If entry is present, then the gateway inserts the entries as follows:
- If the maximum and minimum dropped entries of the packet are empty then, max:PNO_i = max:PNO
  min:PNO_i = min:PNO
  and the corresponding entry is deleted.
- If the maximum and minimum dropped entries of the packet are not empty then, max:PNO_i = max:PNO_i
  min:PNO_i = min:PNO
  and the corresponding entry is deleted.

TCP receiver: Processing data packets In this section, we describe the algorithm that is used to maintain the reorder list. The data packets that arrive at the receiver could bring in the max-min dropped information about any dropped packets irrespective of the sequence. For example, 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. (Note: In the following algorithm, the reordered list and the drop list is searched, compared and deleted only when the list is non-empty. Also when the packets are inserted into the drop list or the reordered list, we make sure that the sequence number of the current packet (PNO_i) and the sequence number of the highest received packet (PNO_k) in the receiver buffer queue are not inserted.) When a data packet $< F_{id}, PNO_i >$ arrives at the TCP receiver,

– The TCP receiver checks whether the dropped entries min:PNO_i and max:PNO_i are empty or not.
  - If those values are null, the TCP receiver checks if the sequence number of the current packet (PNO_i) > the sequence number of the highest received packet (PNO_k) in the receiver buffer queue.
    * If greater, then the TCP receiver checks for gaps between PNO_i and PNO_k AND if those sequence numbers that needs to fill those gaps are present in the drop list.
      · If yes, then the TCP receiver assumes that those packets have been dropped.
      · If no, then the packets within the gaps are probably reordered. The TCP receiver adds them to the reordered list.
    * If lesser, then the TCP receiver checks if PNO_i is in the reordered list. If yes, then the packet is removed from the reordered list.
  - If the values are not null, the TCP receiver checks if PNO_i is in the reordered list. If present, then the entry is deleted. Then the TCP receiver checks if $PNO_i > PNO_k$.
    * If yes,
      · The TCP receiver checks if the min:$PNO_i > PNO_k$ AND max:$PNO_i < PNO_i$. The TCP receiver checks for gaps between min:PNO_i and PNO_k and also checks for gaps between max:PNO_i and PNO_i. If there are gaps, the TCP receiver adds them to the reordered list. Whilst adding, check if the sequence numbers from min:PNO_i to max:PNO_i are present in the reordered list. If present, remove them from the reordered list.
      · If the min:$PNO_i > PNO_k$ AND max:$PNO_i > PNO_i$. The TCP receiver checks for gaps between min:PNO_i and PNO_k. If there are gaps, the TCP receiver adds them to the reordered list. Then the TCP receiver checks if the sequence numbers from min:PNO_i to PNO_i are present in the reordered list. If present, remove them from the reordered list. Put the sequence numbers from $PNO_i + 1$ to max:PNO_i into the drop list for future references.

· If the min:$PNO_i < PNO_k$ AND max:$PNO_i < PNO_i$. The TCP receiver checks for gaps between max:PNO_i and PNO_i, If there are gaps, the TCP receiver adds them to the reordered list. Then the TCP receiver checks if the sequence numbers from min:PNO_i to max:PNO_i are present in the reordered list. If present, remove them from the reordered list.

· If the min:$PNO_i < PNO_k$ AND max:$PNO_i > PNO_i$. The TCP receiver checks if the sequence numbers from min:PNO_i to PNO_i are present in the reordered list. If present, remove them from the reordered list. Put the sequence numbers from $PNO_i + 1$ to max:PNO_i into the drop list for future references.

∗ If no,

· If the min:$PNO_i > PNO_k$ AND max:$PNO_i > PNO_i$. The TCP receiver checks for gaps between min:PNO_i and PNO_k. If there are gaps, the TCP receiver adds them to the reordered list. Put the sequence numbers from min:PNO_i to max:PNO_i into the drop list for future references.

· If the min:$PNO_i < PNO_k$ AND max:$PNO_i > PNO_i$. The TCP receiver checks if the sequence numbers from min:PNO_i to PNO_k in the queue are present in the reordered list. If present, remove them from the reordered list. Put the sequence numbers from PNO_k to max:PNO_i into the drop list for future references.

· If the min:$PNO_i < PNO_k$ AND max:$PNO_i < PNO_i$. The TCP receiver checks for gaps between max:PNO_i and PNO_i. If there are gaps, the TCP receiver adds them to the reordered list. The TCP receiver checks if the sequence numbers from min:PNO_i to max:PNO_i are present in the reordered list. If present, remove them from the reordered list.

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

– Checks if an incoming packet is filling a gap. If yes, check if the packet following the current packet (in sequence number) is in the

reordered list. If yes, the reordered bit is set and the cumulative ACK is sent.
– If the packet does not fill a gap, then the receiver checks whether the sequence number following the last inorder packet is in the reordered list. If yes, the reordered bit is set for that particular SACK packet.

TCP Sender : Processing the acknowledgements When an acknowledgement is received, the TCP sender does the following,

– If none of the three *dupacks* received have their reordered 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 reordered 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 reordered 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 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 packets following the dropped packet in that particular gap (which can be found out by the SACK packet) are dropped and retransmits them after receiving 3 *dupacks*.

3.2 Storage and Computational Costs

The TCP options field has 40 bytes. We use 4 bytes for each minimum and maximum dropped entries to be inserted into the option field of the TCP segment. 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 reordered list depends on the amount of packets the TCP receiver assumes that has been reordered in the network by measuring the gaps. Thus the computational cost involved in Insertion is O(n) where n is the number of missing packets within a gap. Deletion and comparison costs O(m) where m is the length of the reordered 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.

4 Simulation Environment

We use the network simulator ns-2 [11] to test our proposed solution. We created our own version of a reordering gateway and made

changes to the TCP SACK protocol. Assume nodes A and B are the sender and destination nodes respectively and R1, R2 are routers. Nodes A and B are each connected through router R1 and R2 via 10Mbps Ethernet having a delay of 1ms. The routers R1 and R2 are connected to each other via 5Mbps link with a delay of 50ms. Our simulations use 1500 byte segments. The maximum congestion window was 500 segments. The experiments were conducted using a single bulk TCP transfer. The queue size used was 65 segments. We used the drop-tail queuing strategy. We varied the frequency of reordering with respect to time and the number of random queue swaps to determine the throughput.

5 Impact of Reordering

In this section, we compare the throughput performance of the simulated network using TCP SACK, for various levels of reordering. We tested our network to determine the average throughput as a function of random queue swaps performed every 1 or 8 seconds. For e.g. when there was no reordering in the network, the average throughput of a 10 minute TCP connection was 607500 bytes/second. When reordering was introduced say 15 queue swaps every second, the average throughput was 526000 bytes/second. Thus the throughput reduced to about 13.4%. When we increased the amount of swaps every 1 or 8 seconds, the average throughput gradually decreased.

This shows that persistent reordering degrades the throughput performance of a network to a large extent.

6 Results

We performed various tests by varying the value of 'k'. When $k = 0$ ($dupthresh = 3$), the network behaves as an ordinary network with persistent reordering events using TCP. When $k = 1$, the performance of the network was similar to the network with $k = 0$. When $k > 1$, the throughput increases rapidly by reducing the number of unnecessary retransmits that occur due to reordering. In this paper, we present the results for value of $k = 3$ ($dupthresh = 6$).

We compared the throughput performance of the simulated network using RD-TCP for various levels of reordering. We tested our

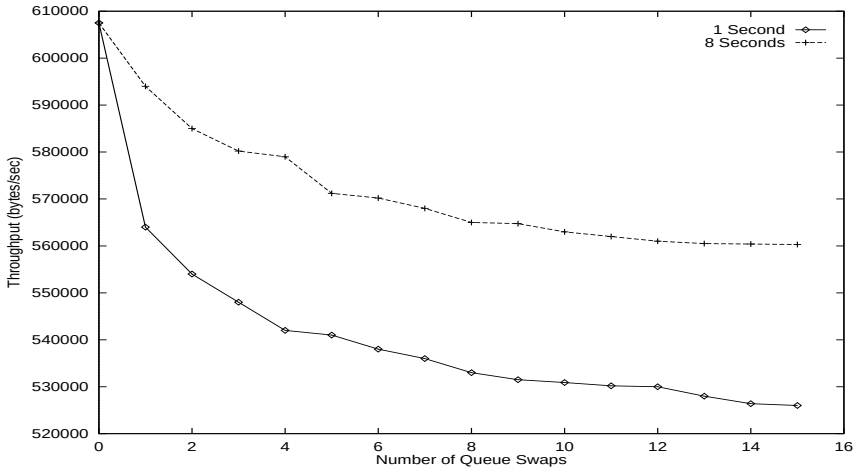

Fig. 1. Comparison of throughput performance of the network using TCP SACK as a function of number of random queue swaps

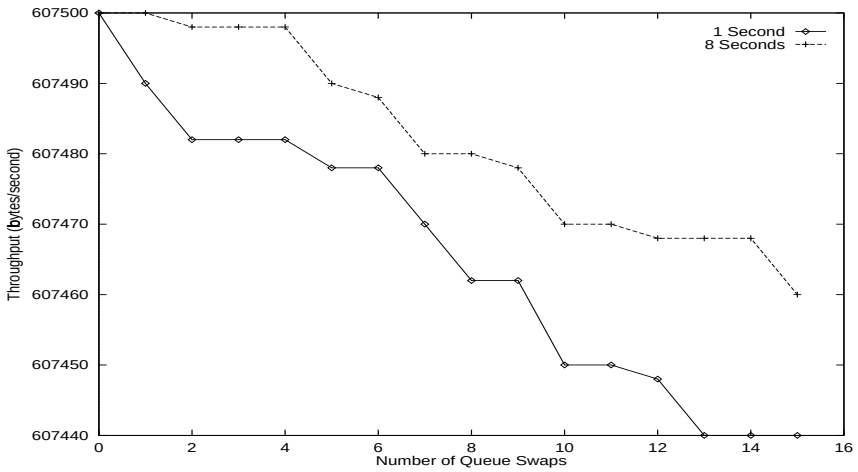

Fig. 2. Comparison of throughput performance of the network using RD-TCP as a function of number of random queue swaps

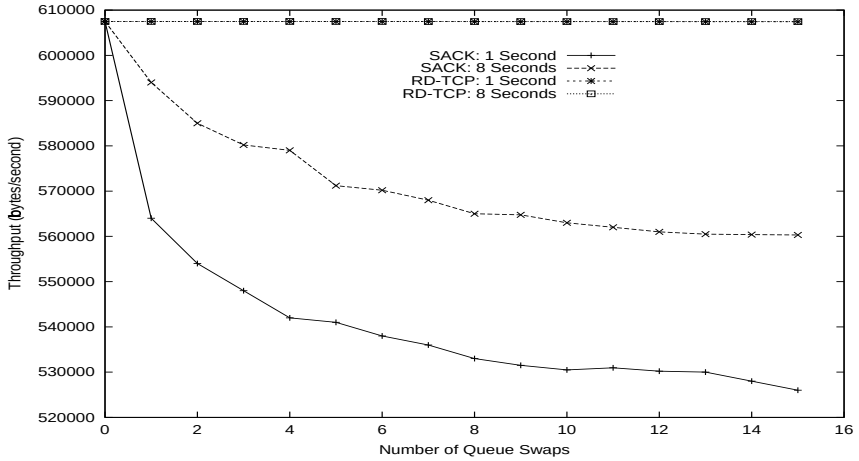

Fig. 3. Comparison of throughput performance of the network using TCP SACK and RD-TCP as a function of number of random queue swaps

network to determine the average throughput as a function of random queue swaps performed every 1 or 8 seconds. Figure 2 shows the throughput (number of data bytes/second) as a function of the number of queue swaps performed every 1 or 8 seconds. When reordering was introduced say 15 queue swaps every second, the average throughput was 607440 bytes/second. Thus the throughput reduced was negligible compared to the 13.4% drop in performance of TCP SACK. Overall, the throughput reduced negligibly when the frequency of reordering was increased. Figure 3 shows the throughput comparison between RD-TCP and TCP SACK.

We also compared the throughput performance of the network with reordering events using TCP DSACK to the same network with reordering events using RD-TCP. Figure 4 shows the comparison of the throughput performance of RD-TCP and TCP using DSACK. When reordering was introduced, RD-TCP performed much better that the TCP using DSACK. For e.g. when the number of queue swaps every second was 15, the average throughput of TCP using DSACK was 600600 bytes/second whereas the average throughput of RD-TCP was only 607440 bytes/second.

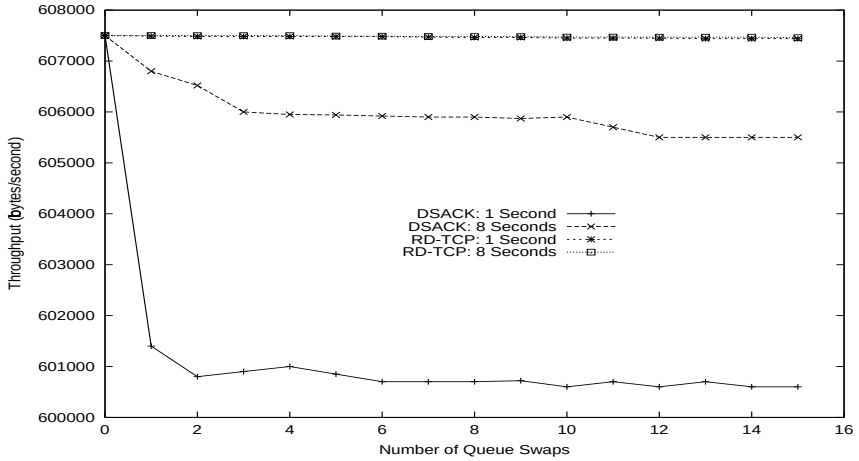

Fig. 4. Comparison of throughput performance of the network using DSACK and RD-TCP as a function of number of random queue swaps

7 Conclusions and Future Work

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

Further work in this area includes:

– Further work needs to be done to the TCP receiver to identify the amount of reordering that has occurred in the network and to inform the TCP sender about this information. The TCP sender can then increase the value of dupthresh by some value 'k' according to the degree of reordering.

– The simulated results presented in this paper needs verification in the real network.
– Further simulations and testing needs to be carried out to find the efficiency of the protocol when there is an incremental deployment i.e. when there are some routers in a network which have not been upgraded to use our mechanism.
– We need to test the performance of RD-TCP for a realistic model of reordering based on delays rather than swapping segments.

References

1. Paxson, V.: End-to-End Routing Behaviour in the Internet. Proceedings of the SIGCOMM (1996)
2. Chinoy, B.: Dynamics of Internet Routing Information. Proceedings of the SIGCOMM (1993)
3. Estan, C., Varghese, G.: New Directions in Traffic Measurement and Accounting. Proceedings of the SIGCOMM (2002)
4. Jacobson, V.: Congestion Avoidance and Control. Proceedings of the SIGCOMM (1988)
5. Ludwig, R., Katz, R.: The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. Computer Communication Review, 30(1)(2000)
6. Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M.: An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (2000)
7. Allman, M., Paxson, V.: On Estimating End-to-End Network Path Properties. Proceedings of the SIGCOMM (1999)
8. Bennett, J., Partridge, C., Shectman, N.: Packet Reordering is Not Pathological Network Behaviour. IEEE/ACM Transactions on Networking (1999)
9. Postel, J.: Transmission Control Protocol. RFC 793 (1981)
10. Blanton, E., Allman, M.: On Making TCP More Robust to Packet Reordering. Proceedings of the SIGCOMM (2002)
11. McCanne, S., Floyd, S.: Network Simulator. http://www.isi.edu/nsnam/ns/
12. Fang, W., Peterson, L.: Inter-as-traffic patterns and their implications. IEEE GLOBECOM (1999)
13. Allman, M., Balakrishnan, H., Floyd, S.: Enhancing TCP's Lost Recovery Using Limited Transmit. RFC 3042 (2001)
14. Cisco NetFlow: http://www.cisco.com/warp/public/732/Tech/netflow
15. Mogul, J.: Observing TCP Dynamics in Real Networks. Proceedings of the SIGCOMM (1992).
16. Sathiaseelan, A., Radzik, T.: Proceedings of the 6th IEEE International Conference on High Speed Networks and Multimedia Communications HSNMC'03, Portugal, July 2003 (LNCS 2720, pp.471-480).