# RD-TCP: Reorder Detecting TCP

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 with transmission that follows predominantly, but not necessarily exclusively, symmetric routing paths, and we 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* [7]: 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 also due to multi-path routing. 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 [8]. 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 [8].

M.M. Freire, P. Lorenz, M.M.-O. Lee (Eds.): HSNMC 2003, LNCS 2720, pp. 471–480, 2003.
© Springer-Verlag Berlin Heidelberg 2003

*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 [3]. 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 [9]:

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 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 indication about the extent of the deterioration of the TCP performance when reordering of packets occurs, and the summary results from some of our simulations are shown in Table 1, columns 1–4 (we present the details of our simulations in Sections 4 and 5). For example, when the gateways were of the *Drop-tail* type and the queue sizes were 65, we observed that the TCP throughput decreased by 18% when reordering of packets occurred (Table 1, the first row).

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. 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).

RD-TCP should perform better than the standard TCP, if reordering of packets commonly occurs and if the senders receive confirmation, via the bits

**Table 1.** Normalized TCP throughput in an example network

| *type of gateways* | *queue size* | standard TCP, no reordering | standard TCP, reordering | RD-TCP, reordering |
|---|---|---|---|---|
| Drop-tail | 65 | 1.00 | 0.82 | 0.98 |
| Drop-tail | 20 | 1.00 | 0.87 | 1.03 |
| RED | 65 | 1.00 | 0.86 | 0.98 |
| RED | 20 | 1.00 | 0.88 | 0.98 |

set in *acks*, for the large proportion of dropped packets (it is not necessary that the senders receive confirmation for all dropped packets). Thus the performance of RD-TCP should be better than the performance of the standard TCP for networks, for which the following three conditions are true. Conditions 2 and 3 together insure that the senders are notified about most of the dropped packets.

1. Reordering of packets is common.
2. Large proportion of routing is symmetrical, that is, the acknowledgement packet is sent along the same path followed by the data packet.
3. Large proportion of gateways record information of dropped packets.

Paxson's study on the Internet [1] shows that approximately half of the measured routes are symmetrical, but for the local area networks this proportion should be considerably higher. The performance of RD-TCP which we observed in our simulations, are summarized in Table 1, column 5. Thus in our simulations RD-TCP performed significantly better than the standard TCP. In fact, it performed almost as well as if no reordering of packets occurred.

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 short discussion of the further research in Section 7 and a summary of our work in Section 8.

## 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 [4].
- 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].
- A method has been proposed by [6], 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.

– [9] proposes various techniques for changing the way TCP senders decide to retransmit data segments by estimating the amount of reordering in the network path and increasing *dupthresh* by some value K, whenever a spurious fast retransmit occurs.

These methods show ways of improving the TCP performance when a packet has been retransmitted in the event of reordering. In our paper, we try to improve the performance by 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 in the gateways, by having a data structure that maintains the sequence number of the packet that gets dropped in the gateway. When an *ack* for some data packet $P_k$ arrives at the gateway, the data structure is searched to check whether the sequence number of the packet $P_{k+1}$ has been dropped by that particular gateway or not. If the packet has been dropped, then a *dropped* bit is set in the *ack*. When the sender receives an *ack*, it checks for the *dropped* bit and if it is set, then the sender knows that the packet has been dropped and retransmits the lost packet after receiving three *dupacks*. If the *dropped* bit is not set, then the TCP sender assumes that the packet has been reordered in the network and waits for 'k' more *dupacks* ('3+k' in total) instead of three *dupacks* to resend the data packet. 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 $P_1, P_2, P_3$ to node B through the gateways R1 and R2. If R1 drops packet $P_2$ due to congestion in the network, then node B will not receive $P_2$. On receipt of packet $P_3$, node B sends a *dupack* (each having sequence number $P_1$) through the gateways R2 and R1. Node A receives this *dupack* assuming the routing is purely symmetrical. Now in our proposed solution, when R1 drops packet $P_2$, the sequence number of packet $P_2$ is inserted into our data structure at R1, which in our case is a hashtable. Node B will not receive packet $P_2$. When node B receives packet $P_3$, it sends a *dupack* When gateway R2 receives an *ack* (having sequence number $P_1$), it checks whether the sequence number for packet $P_2$ ($P_{1+1}$) is available in its data structure. Since R2 does not have an entry, it does not set the *dropped*

bit. When gateway R1 receives the *ack*, it checks for the sequence number for packet $P_2$, and finds that the sequence number is present in the data structure. Gateway R2 then sets the *dropped* bit in the *ack*, meaning that the packet has been dropped by the gateway. When the sender receives three *dupacks* with the *dropped* bit set, the sender retransmits the dropped packet.

Suppose the packet $P_2$ had been reordered in the gateway, the receiver B assuming the packet has been dropped, sends a *dupack* on receipt of packet $P_3$. When gateway R2 receives the *ack*, it checks for the sequence number entry in its hashtable and finds that there is no entry for it, and does not set the *dropped* bit. Similarly when gateway R1 receives the *ack*, it checks for the sequence number in its hashtable and finds that there is no entry for it, and does not set the *dropped* bit. When the sender node A receives a *dupack*, it checks for the *dropped* bit of each of these 2 *acks*, and when '3+k' *dupacks* with the *dropped* bit not set are received, the packet is resent and fast recovery is triggered. If the value of 'k' is not large enough, then the 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 [9] 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.

### 3.1 Details of the Implementation

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

**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, then sequence number of the dropped packet ($PNO_i$) is inserted into the end of the list of the corresponding flow id. If an entry is not present, an entry is created, and the sequence number of the dropped packet is entered as the first entry in the list.

**Processing the *ack* Packets** When an $ack < F_{id}, PNO_i >$ arrives at the gateway,

- If the *dropped* bit is already set (some other gateway has dropped the packet), then pass on the packet.

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

**Removing Inactive Lists** There may be cases where possible residuals (packet sequence numbers) may be left in the list even though that particular flow has become inactive. To remove these unwanted residuals and the list for that particular flowid, we could have another hash table that maintains a timestamp of the last *ack* packet that has passed through the gateway for the flow whose entry is already present in the main hash table (When an entry is created in the main hash table for a particular flow, an entry is also created in this hashtable simultaneously). It uses the flow id ($F_{id}$) as the index of the hashtable. The timestamp entry for that particular flow is regularly updated with the timestamp of the last *ack* that has passed through the gateway. Regularly, say every 300 ms, the entire hashtable is scanned, and the difference of the timestamp entry in each index with the current time is calculated. If the difference is greater than a set threshold (say, 300ms), then it means that the flow is inactive currently and the entries of both the hashtables for that particular flow are removed.

### 3.2 Storage and Computational Costs

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. Despite the large number of flows, a common observation found in many measurement studies is that a small percentage of flow accounts for a large percentage of the traffic. It is argued in [11] that 9% of the flows between AS pairs account for the 90% of the byte traffic between all AS pairs. It is shown in [2] that large flows could be tracked or monitored easily by using SRAM that copes up with the link speed. Our monitoring process records only flows whose packets have been dropped. To get some rough estimate of the amount of memory needed for our implementation, let us assume that there are $200,000$ concurrent flows passing through one gateway, 10% of them have information about one or more dropped packets recorded in this gateway, and a non-empty list of sequence numbers of dropped packets has on average 10 entries. Thus the hash table will have $20,000$ non-empty entries and the total length of the lists of sequence numbers of dropped packets will be $200,000$. We need 4 bytes for each flow id, 4 bytes for each packet sequence number, and another 4 bytes for

each pointer. This means that the total memory required would be about 2.5 MB. This is only a rough estimate of the amount of extra memory needed, but we believe that it is realistic. Thus we expect that an extra 8MB SRAM would be highly sufficient to implement our solution.

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

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 [10] to test our proposed solution. We created our own version of a reordering gateway and made minor changes to the TCP protocol. Assume nodes A and B are the sender nodes, nodes C and D are the destination nodes and R1, R2 are routers. Nodes A and B are each connected to router R1 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 10ms. Nodes C and D are each connected to router R2 via 10Mbps Ethernet having a delay of 1ms. Our simulations use 1500 byte segments. We have conducted experiments with both Drop-tail and RED gateways. The queue sizes used were 20 and 65 segments. In our experiments we have used FTP traffic flows between source node A and destination node C via routers R1, R2 and between source node B and destination node D. For Drop-tail queues, we have simulated an average of 6 reorder events per second and the pattern of reordering is consistent through out the experiments using Drop-tail queues. For RED queues, we have simulated an average of 4 reorder events per second and the pattern of reordering is consistent throughout the experiments using RED queues.

## 5 Impact of Reordering

In this section, we compare the throughput performance of the simulated network using TCP, with and without reordering events for both Drop-tail and RED gateways each having a queue size of 65 segments. The reason why we chose a queue size of 65 was to prevent packet drops so that we can easily verify the impact of reordering on a network where there is no packet drops. When reordering occurred in Drop-tail gateways, there was a 17.9% reduction

in throughput performance when compared to the throughput performance of the network without any reordering events (refer Table 2 (first row, columns 2 and 3)). Similarly when reordering occurred in RED gateways, there was a 13.8% reduction in throughput performance when compared to the throughput performance of a network without any reordering events (refer Table 3 (first row, columns 2 and 3)). 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 is 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. We have summarized the results for $k = 2$ (*dupthresh* = 5) in Tables 2 and 3.

| *queue size* | standard TCP, no reordering | standard TCP, reordering occurs | RD-TCP, reordering occurs |
|---|---|---|---|
| 65 | 6144680 | 5042040 (-17.9%) | 6007620 (-2.2%) |
| 20 | 5787400 | 5042040 (-12.8%) | 5867480 (+1.0%) |

**Table 2.** Comparison of throughput performance of the network using Drop-tail gateways for different queue size.

We compared the throughput performance of the network with reordering events using TCP to the same network with reordering events using RD-TCP with a Drop-tail queue size of 65. Figure 1 shows the comparison of the throughput performance of RD-TCP and TCP using Drop-tail queues. The total number of bytes received at the end of the 10 minute simulation by the network with reordering using TCP was 5042040 bytes, whereas the total bytes received at the end of the 10 minute simulation by the network with reordering using RD-TCP was 6007620 bytes. Interestingly we achieved 19.1% increase in throughput performance when compared to the same network with persistent reordering using TCP. There was only a 2.2% reduction in throughput performance when compared to the throughput performance of the network without any reordering events using TCP (refer Table 2 (first row, columns 2 and 4)).

We compared the throughput performance of the network with reordering events using TCP to the same network with reordering using RD-TCP with a RED queue size of 65. The total number of bytes received by the two receivers at the end of the 10 minute simulation by the network with reordering using TCP was 5294600 bytes, whereas the total bytes received by the two receivers at the end of the 10 minute simulation by the network using reordering with RD-TCP

| *queue size* | standard TCP, no reordering | standard TCP, reordering occurs | RD-TCP, reordering occurs |
|---|---|---|---|
| 65 | 6144680 | 5294600 (-13.8%) | 6016860 (-2.1%) |
| 20 | 6016860 | 5308460 (-11.7%) | 5898280 (-1.9%) |

**Table 3.** Comparison of throughput performance of the network using RED gateways for different queue size.

was 6016860 bytes. Interestingly we achieved 13.64% increase in throughput performance when compared to the same network with persistent reordering using TCP. There was only a 2.1% reduction in throughput performance when compared to the throughput performance of the network without any reordering events using TCP (refer Table 3 (first row, columns 2 and 4)).

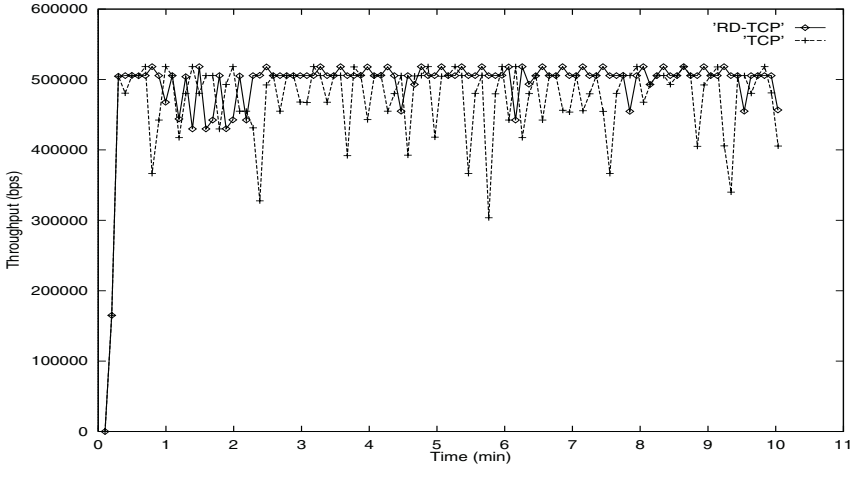


**Fig. 1.** Comparison of throughput performance of the network with reordering events using RD-TCP with k = 2 vs the same network with reordering events using TCP with a Drop-tail queue size of 65.

## 7 Further Work

– We have proposed a mechanism for enabling the senders to distinguish whether a packet has been lost or reordered in the network. We have initially simulated and tested our protocol on networks whose routing are symmetric. Further work has to be done to test the efficiency of our protocol for various levels of asymmetric routing.

- Our proposed solution should be compared with the other solutions that have been mentioned in the related work section. These methods show ways of improving the TCP performance when a packet has been retransmitted in the event of reordering.
- 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.

## 8 Conclusion

In this paper, we proposed a solution that prevents the unnecessary retransmits that occur due to reordering events in networks that follow symmetrical routing paths, 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. We have also showed that our solution improves the throughput performance of the network to a large extent.

## References

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