Technische Universität Berlin

Institut für Telekommunikationssysteme
Internet Network Architectures

Fakultät IV
FG INET/MAR 4-4
Marchstr, 23
10587 Berlin
http://www.inet.tu-berlin.de

Master Thesis

Investigating a reordering robust TCP for storage cluster and data center use.

Puneeth Nanjundaswamy

Matriculation Number: 0359759
01.06.2015

Supervised by:
Prof. Dr. Anja Feldmann, Ph.D
Dr. Lars Eggert, Ph.D
Dr. Alexander Zimmermann, Ph.D

NetApp Germany GmbH
Sonnenallee 1
85551 Kirchheim bei München

This dissertation originated in cooperation with the NetApp Germany GmbH.

I would like to take this opportunity to thank my supervisor at NetApp, Dr. Lars Eggert for providing me with the wonderful opportunity to work on my master thesis at NetApp, Munich and for all the great support. It was a wonderful experience working with you and I thoroughly enjoyed it. I would also like to thank Dr. Alexander Zimmermann for providing me with all the support and motivation while working on my master thesis. Furthermore, I would also like to thank Dr. Douglas Santry for all the support and helpful advices during my stint at NetApp.

Acknowledgement

I would like to thank Prof. Anja Feldmann for providing me with the opportunity to work on my master thesis at INET, TU Berlin. I would also like to thank Dr. Prométhé Spathis at UPMC Paris, Nina Reinecke at TU Berlin and all the members of EIT ICT Labs for their continuous support during my Masters program.

I would like to dedicate this work to my family for their continuous unwavering support, my best friends Sunil C Ramanarayanan, Nakul Ganesh for being there and helping me and the entire Erasmus group from Paris.

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, 01.06.2015

. . . .

(Signature Puneeth Nanjundaswamy)

Abstract

Transmission control protocol (TCP) is a highly reliable host-to-host protocol which ensures reliable transfer of data over unreliable internet. TCP must recover from damaged, lost, duplicated and out-of-order data. TCP sender retransmits a data segment upon receiving three duplicate acknowledgements(ACK) from the receiver. Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for 3 duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK.

However, the above assumption is not valid any more as numerous studies in the wild internet have found that packet reordering is not rare and that waiting for three duplicate ACKs might not be sufficient to disambiguate packet reordering and packet loss. Retransmission of data segments also results in the decrease of the sender's throughput. Thus an efficient reordering detection and reaction mechanism is needed to avoid spurious retransmissions. Avoiding spurious retransmissions reduces the unnecessary decrease in the sender's throughput, there by improving the performance of TCP.

This thesis describes packet reordering and its metrics in detail. Furthermore, it studies the performance impacts of packet reordering on TCP. It then investigates the various approaches by which TCP can be made robust against packet reordering with focus on the Linux's implementation, Non congestion to robustness (NCR) and adaptive NCR algorithm (aNCR) of detecting and handling packet reordering. The algorithms are implemented in the Linux kernel and evaluated under various conditions of packet reordering and under multiple traffic conditions to determine the best performing algorithm under all scenarios.

Zusammenfassung

TCP (Transmission control protocol) ist ein Datentransportprotokoll, das Daten zuverlässlich durch das unzuverlässliche Internet übermittelt. TCP musst sich erholen können von beschädigten, verlorenen, duplizierten Daten und von Daten, die in falscher Reihenfolge auftreten. Der TCP-Sender schickt einen Datensegment zum zweiten Mal, wenn vom Empfänger 3 Duplikat-Meldungen (duplicate acknowledgement, kurz ACK) gekommen sind. DA TCP nicht erkennen kann, ob ein ACK durch ein verlorenes Segment oder durch einer falschen Reihenfolge von Segmenten verursacht wird, wartet es auf drei ACKs. Angenommen wird, dass es im Falle einer Umordnung von Segmenten nur zu ein oder zwei Duplikat-ACKs kommt bevor das verschobene Segment verarbeitet wird, was wiederum ein neues ACK generiert.

Die obige Annahme ist jedoch nicht mehr gültig, widerlegt durch mehrere Studien, die rausgefunden haben, dass Paketumordnung keine seltene Erscheinung ist, und daher das Warten auf nur drei Duplikat-ACKs nicht ausreicht um zwischen Paketverlust und Paketumordnung zu unterscheiden. Neuübermittlung von Datensegmenten resultiert auch in der Verringerung des Senders Durchsatzes. Ergo wird eine effiziente Methode zur Erkennung Verarbeitung von Neuordnung benötigt, um obsolete Wiederversenden von Daten zu vermeiden. Gelingt dies, wird der Durchsatz, und damit die Leistung des TCP erheblich verbessert.

Diese Arbeit beschreibt detailliert Paketumordnung und deren Metrik. Darüberhinaus werden Auswirkungen auf die Leistung durch Paketumordnungen studiert. Verschiedene Lösungsansätze werden dann auf ihre Fähigkeit untersucht, TCP robuster gegen Paketumordnung zu machen, mit Fokus auf die Linuximplementation, NCR- und aNCR-Algorithmen für die Erkennung und Verarbeitung von Paketumordnung. Die für den Linux kernel Implementierten Algorithmen werden unter verschiedene Bedingungen auf Paketumordnung und Datenverkehr ausgewertet, um den am besten arbeitenden Algorithmus zu bestimmen.

Contents

List of Figures xv

Listings xvii

List of Tables xix

Definitions xxi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Thesis outline . . . 3

2 Background and related work 5
- 2.1 Packet reordering . . . 5
  - 2.1.1 Reordering extent . . . 6
- 2.2 Packet reordering in the internet . . . 6
- 2.3 TCP extensions . . . 8
  - 2.3.1 TCP selective acknowledgement (SACK) option . . . 8
  - 2.3.2 TCP duplicate selective acknowledgement (DSACK) option . . . 8
  - 2.3.3 TCP timestamp option . . . 9
  - 2.3.4 Deployment trends of TCP options . . . 9
- 2.4 Impact of packet reordering on TCP . . . 9
  - 2.4.1 Forward path reordering . . . 10
  - 2.4.2 Reverse path reordering . . . 11
  - 2.4.3 Forward and reverse path reordering . . . 12
- 2.5 Impact on future technologies . . . 12
- 2.6 Approaches to make TCP robust to packet reordering . . . 12

3 TCP-NCR and TCP-aNCR 15
- 3.1 Extended limited transmit (ELT) . . . 15
  - 3.1.1 Careful limited transmit (CF ELT) . . . 15
  - 3.1.2 Aggresive limited transmit (AG ELT) . . . 16
  - 3.1.3 Hybrid limited transmit (HY ELT) . . . 17
- 3.2 TCP-NCR . . . 18
  - 3.2.1 Algorithm . . . 18
  - 3.2.2 Drawbacks of TCP-NCR . . . 21

3.3 TCP-adaptive NCR 22
3.3.1 Reordering detection 22
3.3.2 Reordering reaction 26

4 Measurement Setup 33
4.1 Testbed 33
4.2 Traffic shaping and policing 35
4.3 Packet reordering in Linux kernel 36
4.4 Pluggable framework for reordering algorithms 37
4.5 Flowgrind: Network performance tool 38

5 Evaluation and Discussion 41
5.1 Lower data rates 41
5.1.1 Scenario 1: Performance without packet reordering under different bottleneck bandwidths 41
5.1.2 Scenario 2: Performance with packet reordering under different bottlenecks 44
5.1.3 Scenario 3: Performance with packet reordering under varying RTTs 48
5.1.4 Scenario 4: Performance with varying reordering rate 48
5.1.5 Scenario 5: Performance with varying reordering delay 52
5.1.6 Scenario 6: Performance with reverse path loss 56
5.2 Data center environment 58
5.2.1 Scenario 1: Performance without packet reordering. 59
5.2.2 Scenario 2: Performance under mild packet reordering. 59
5.2.3 Scenario 3: Performance under varying reordering rates. 60

6 Conclusion and future work 63
6.1 Summary 63
6.2 Future work 63

Bibliography 65

List of Figures

1.1 TCP: Fast retransmit 2
1.2 Packet reordering and Reordering extent 3

2.1 Packet reordering and Reordering extent 7
2.2 TCP congestion control and loss recovery 13

3.1 Careful limited transmit 16
3.2 Aggresive limited transmit 17

4.1 Testbed for evaluating under low data rates 34
4.2 Testbed for evaluating under higher data rates 35
4.3 A sample flowgrind logfile output 39

5.1 Lower data rate - Scenario 1: Average throughput vs BNBW 43
5.2 Lower data rate - Scenario 2: Average throughput vs BNBW 47
5.3 Lower Data rate - Scenario 3: Average throughput vs delay 49
5.4 Lower data rate - Scenario 3: Average application-perceived latency vs delay 50
5.5 Lower data rate - Scenario 3: Spurious retransmits at various RTT values 51
5.6 Lower data rate - Scenario 4: Average throughput vs Reordering rate . . 53
5.7 Lower data rate - Scenario 4: Spurious retransmits at various RTT values 54
5.8 Lower data rate - Scenario 5: Average throughput vs Reordering delay . . 55
5.9 Lower data rate - Scenario 5: Spurious retransmits at various reordering delays 56
5.10 Lower data rate - Scenario 6: Average throughput vs Reordering delay . . 57
5.11 High data rate: Performance of algorithms under no packet reordering . . 59
5.12 High data rate: Performance of algorithms under mild packet reordering . 60
5.13 High data rate - scenario 3: Average throughput at various reordering rates 61

Listings

4.1 Testbed sysctl settings . . . 34
4.2 Configuring Linux kernel for NetEm . . . 35
4.3 Setting up basic HTB bucket on a node . . . 36
4.4 Setting up a filter for the HTB bucket . . . 36
4.5 Setting up a delay for the HTB bucket . . . 36
4.6 Packet reordering with gap . . . 37
4.7 Packet reordering with correlation . . . 37

List of Tables

5.1 Scenario 1: Average application-perceived latency (seconds) vs BNBW (mbps) - Reno 44
5.2 Scenario 1: Average transactions/s vs bottleneck bandwidth (mbps) - CUBIC 45
5.3 Scenario 2: Average transactions/s vs bottleneck bandwidth (mbps) - CUBIC 46
5.4 Scenario 4: Average transactions/s vs reordering rate - CUBIC 52
5.5 Scenario 5: Average transactions/s vs reordering delay - CUBIC 56

Definitions

Segment: A segment is any TCP/IP data or acknowledgment packet (or both).

Sender maximum segment size (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [RFC1191, RFC4821] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

Receiver maximum segment size (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, it is 536 bytes [RFC1122]. The size does not include the TCP/IP headers and options.

Full sized-segment: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

Receiver window (rwnd): The most recently advertised receiver window.

Congestion window (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

Initial window (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

Loss window (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.

Restart window (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle period (if the slow start algorithm is used; see section 4.1 for more discussion).

Flight size: The amount of data that has been sent but not yet cumulatively acknowledged.

Duplicate acknowledgement: An acknowledgement is considered a "duplicate" in the

following algorithms when (a) the receiver of the ACK has outstanding data, (b) the incoming acknowledgment carries no data, (c) the SYN and FIN bits are both off, (d) the acknowledgement number is equal to the greatest acknowledgement received on the given connection (TCP.UNA from [RFC793]) and (e) the advertised window in the incoming acknowledgement equals the advertised window in the last incoming acknowledgement.

Alternatively, a TCP that utilizes selective acknowledgements (SACKs) [RFC2018, RFC2883] can leverage the SACK information to determine when an incoming ACK is a "duplicate" (e.g., if the ACK contains previously unknown SACK information).

HighACK: It is the sequence number of the highest byte of data that has been cumulatively ACKed at a given point.

HighData: is the highest sequence number transmitted at a given point.

HighRxt: is the highest sequence number which has been retransmitted during the current loss recovery phase.

Pipe: is a sender's estimate of the number of bytes outstanding in the network. This is used during recovery for limiting the sender's sending rate. The pipe variable allows TCP to use a fundamentally different congestion control than specified in [RFC2581]. The algorithm is often referred to as the "pipe algorithm".

1 Introduction

1.1 Motivation

Transmission Control Protocol (TCP)[1] is a highly reliable host-to-host protocol in a packet switched computer network. one of the main responsibilities of TCP is to provide reliability and should be able to recover from damaged, lost, duplicated or out-of-order packets.

TCP has two mechanisms to determine if a packet lost. The first mechanism is maintaining a retransmission timer which is used if the receiver does not acknowledge the packet before a timeout. The second mechanism is the ACK clocking by the receiver. For each packet received by the receiver, it sends an acknowledgement to the sender. If an expected packet (to ensure in-order delivery to the application) is not received by the receiver, it sends an acknowledgement with the sequence number which was cumulatively acknowledged. Upon the receipt of three such duplicate acknowledgements (DUPACKs), the sender retransmits the data which is assumed lost. The duplicate threshold duplicate threshold (DUPTHRESH) provides a means of robustness against small occurrences of packet reordering.

However today, DUPTHRESH of three is not enough and does not make TCP robust against packet reordering. In fig 1.1 we see that the sender upon receiving three DUPACKs, retransmits data as it assumes it to be lost. However it is spurious retransmission as the original packet was not lost but in fact reordered by an extent of more than three packets as in fig 1.2. In this figure, the packet was reordered by an extent of 9 packets also commonly referred to as the *"Absolute Reordering extent"* or *"Reordering extent"*. The immediate effect of this spurious retransmission is the reduction in transmission speed by the sender assuming that the network is congested. This makes TCP susceptible to packet reordering and impacts its performance. There are various other effects of packet reordering described in detail in section 2.4. Packet reordering in the internet today is not rare any more. Numerous studies have shown that packet reordering is not pathological as studied in detail in section 2.2. Furthermore, it brings about a constraint in the design of future technologies. Thus there is a dire need of making TCP robust against packet reordering.

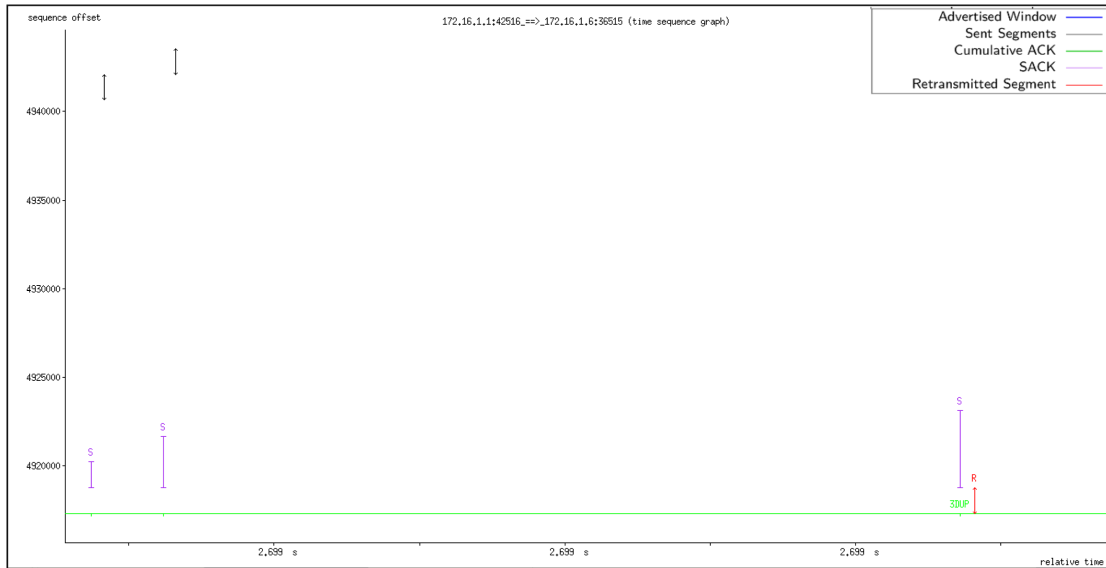


Figure 1.1: TCP: Fast retransmit

1.2 Contributions

The scope of this thesis is to investigate the possible opportunities if TCP is robust against packet reordering and study the performance impact on TCP if it is not robust against packet reordering. Furthermore, the behaviour of Linux, TCP-non congestion to robustness (NCR)[2] and TCP-adaptive non congestion to robustness (aNCR) [3] [4] against packet reordering are studied.

The above algorithms have been implemented in the Linux kernel v3.16 and the performance studied under lower and higher data rates with focus on throughput and application perceived latency for bulk traffic and maximum transaction rate with request-response traffic.

As part of this thesis, a pluggable reordering algorithm framework has been created to test various reordering algorithms under a single kernel. NCR [2] and aNCR [3, 4] algorithms have been implemented as pluggable algorithms in this framework in the Linux kernel version v3.16 and checked for correctness. It has been described in detail in section 4.4.

A new Extended limited transmit *"Hybrid limited transmit"* (section 3.1.3) has been implemented in the Linux kernel making the extended limited transmit compatible with various congestion control algorithms.

Furthermore, Linux's NetEm [5, 6] has been modified to perform advanced packet reordering in our testbed. It has been explained in detail in section 4.3

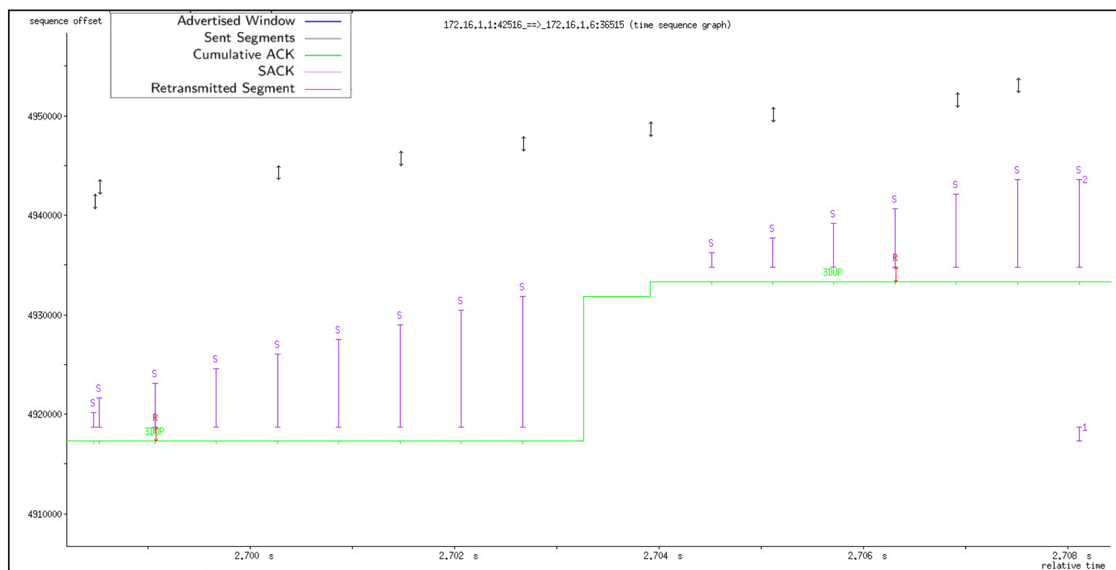


Figure 1.2: Packet reordering and Reordering extent

Finally, Flowgrind [7] has been modified to display more TCP level statistics pertaining to packet reordering such as previously seen *"reordering extent", "total number of spurious retransmissions", "total number of fast retransmits" and "total number of retransmission timeout (RTO) retransmits"*.

1.3 Thesis outline

This Master thesis is separated into 7 chapters and has been structured as follows.

Chapter 2: Background and related work In this chapter, we discuss in detail about packet reordering, its occurance in the wild internet. The TCP extensions that help us determine if a packet is lost or reordered, the deployment trends of these TCP options, the different types of packet reordering in the network and the Linux congestion control state diagram to have an overall view of where the reordering algorithms come into place.

Chapter 3: TCP-NCR and TCP-aNCR This chapter discusses the TCP-NCR and TCP-aNCR algorithm in greater depths. In this section we discuss the advantages and pitfalls of TCP-NCR. Furthermore we discuss how TCP-aNCR overcomes the various pitfalls found in NCR. In this chapter, we also introduce the topic "Extended limited transmit"

Chapter 4: Measurement setup This chapter describes all aspects of the testbed

used for the measurements. We discuss the hardware specifications of the testbed, traffic shaping/policing used, Linux kernel features enabled/disabled, a note on generating packet reordering in the network, description of our Linux kernel with the reordering algorithms and the network performance measurement tool Flowgrind.

Chapter 5: Evaluation and discussion This chapter describes the various test scenarios under which we evaluate the algorithms. The algorithms are tested under various network parameters and under bulk and request-response traffic.

Chapter 6: Conclusion and future work This chapter summarizes the master thesis, describes the problems that occurred during the implementation and an outlook about future work.

2 Background and related work

This chapter talks about the various background information and related work to make TCP more robust to packet reordering. It is assumed that the readers are aware of TCP and thus only discusses packet reordering related work. Firstly section 2.1 describes what is packet reordering, the common metric used to describe packet reordering and the immediate ill effects. The section 2.2 is a brief literature survey about the nature and occurrence of packet reordering in the wild internet. Here we refer to previous studies on packet reordering in the wild internet. In section 2.3, we discuss the various TCP extensions that can be used to detect and react to packet reordering. Furthermore, we also investigate the extent of TCP extensions actually deployed in the wild internet. This exercise gives us a fair idea on the effectiveness of deploying new reordering algorithms which make use of the TCP options. In section 2.4 we discuss the impact of packet reordering on TCP in detail and a note on the possibilities of a reordering robust TCP on future technologies are teased upon in section 2.5. Finally, in section 2.6 we discuss the state machine of TCP congestion control and loss recovery and the approaches to making TCP robust against packet reordering.

2.1 Packet reordering

As cited by RFC 793[1], one of the main purposes of TCP is to provide reliability in computer communication. Reliability against data being damaged, lost, duplicated or delivered out-of-order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring an acknowledgement to each octet received. At the receiver, the sequence number on these octets are used to correctly order the packets and deliver the data to the application layer. If the ACK is not received within a timeout interval, the data is retransmitted. Thus, TCP ensures an in-order-delivery to the application layer.

TCP receiver sends a duplicate ACK for every data received which is not the expected sequence number. TCP does not wait for a retransmit time-out to occur to retransmit the data. Rather, as RFC 2001[8] describes, If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. It is often referred to as Fast Retransmit. During Fast Retransmit, the slow start threshold (ssthresh) is set to half of the current congestion window and the congestion window is set to this new value of ssthresh plus three times the segment size to account for the 3 new data segments sent during the three duplicate

acknowledgements. TCP sender effectively reduces the sending rate to half each time a segment is retransmitted.

The main assumption here is, *It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK.* However, in today's world, this heuristic is not always right. A packet displaced by 4 or more segment numbers is considered lost and not as reordered. This has many negative implications on the performance. In this study[9] the negative implications of false retransmissions on TCP due to packet reordering in the network. It concludes that the throughput is indirectly proportional to packet reordering. At a certain point in time when reordering is high, they find that reordering almost always causes a spurious fast retransmit and halving of the congestion window.[10] further stresses the ill-effects of packet reordering and the TCP behaviour.

The Packet Reordering Metrics RFC 4737[11] describes the various packet reordering metrics to evaluate whether a network has maintained packet order on a packet-by-packet basis. One important metric amongst the many is called *Reordering Extent*.

2.1.1 Reordering extent

Reordering extent is the extent to which packets are reordered and associates a specific sequence discontinuity with each reordered packets. It can be defined as the maximum distance, in packets, from a reordered packet to the earliest packet received that has a larger sequence number. If a packet is in-order, its reordering extent is undefined. The first packet to arrive is in-order by definition and has undefined reordering extent.

The fig 2.1 illustrates the packet reordering phenomenon and reordering extent. The packet number *2* is reordered. The reordering extent of this packet as defined by RFC 4737[11] is *2*.

2.2 Packet reordering in the internet

There have been many studies done to understand the phenomenon of packet reordering and the occurrence of packet reordering in the wild internet. One of the earliest papers to counter the popular belief that packet reordering in the internet is pathological[12] concludes that, though route fluttering, router pauses and broken TCP/IP implementations are cause for packet reordering, a much bigger cause can be attributed to parallelism in links and internet components. In their research, they find that a multiport FDDI switch's feature which allows a collection of ports to operate as a single virtual link to be one of the main the reasons for packet reordering. This research suggests means of countering packet reordering, Firstly, a work around design implementation of this feature in the switch. Secondly, an IP which is aware of the existence of parallel flows and directs traffic between a particular source destination pair always on the same link;

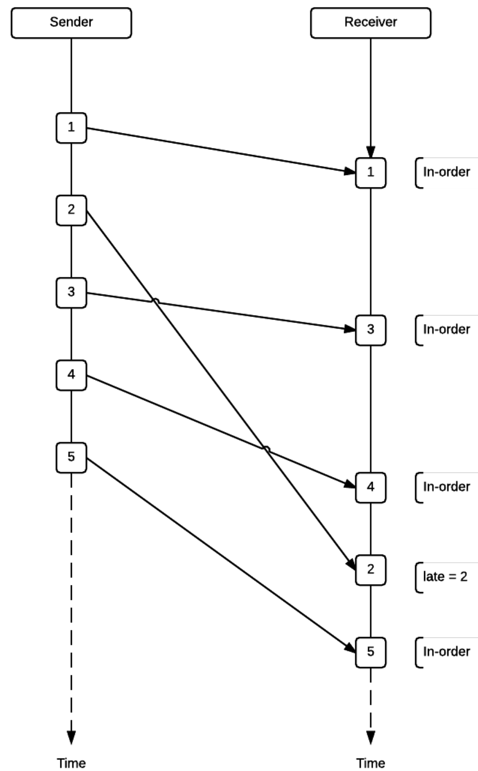


Figure 2.1: Packet reordering and Reordering extent

a TCP with a variable duplicate threshold value. Finally packet numbering on parallel paths. Furthermore, this study also lists the ill-effects of *forward path* and *reverse path* packet reordering on the performance of TCP.

This study on packet dynamics of bulk transfers[13] describes the characteristics of unusual behaviours such as packet reordering and packet replication. They find that packet reordering is prevalent and in their 1st run of experiments, 36% of the connections saw reordering events(data and ACK). They also find that 2% of all the data packets and 0.6% of all all the ACKs from all the connections were reordered. This study as well proposes a TCP scheme where the duplicate threshold can be increased as one of the ways of tackling the ill-effects of packet reordering.

While the previous study focussed on bulk traffic, this research[14] describes the extent of out-of-order packets in HTTP traffic in the wild. Their research is carried out over a three week period on 10,647 Internet web sites in China. They find that in 208 thousand connections with a total of 3.3 million data packets, 3.2% of all the packets were reordered.

This research[15] aims at measuring and classifying packet reordering in backbone networks into retransmissions, network duplication and reordering due to parallelism. The measurements are spread across multiple networks such as content delivery networks (CDN) where traffic originates from clients who are usually closer to the servers, Tier-1 and Tier-2 ISP links which carry multitude of diverse data. They find approximately 1.6% and 5% of out-of-order data in CDN networks and ISP networks respectively. They find approximately 20% of this out-of-order data to be because of spurious retransmissions and another 20% due to network reordering due to parallelism in the network. This study thus emphasizes the need for a reordering robust TCP.

From the previous work, we see the need of a reordering robust TCP;[16] further demonstrates the prevalent packet reordering due to increased parallelism in modern networks, and emphasizes that new application and protocol designs should treat packet reordering on an equal footing to packet loss, and must be robust and resilient to both in order to achieve high performance.

2.3 TCP extensions

There are a number of TCP options available today to make TCP more robust all kinds of network abnormalities.[17][18]. This sections highlights some TCP options that are used in this thesis work to make TCP more robust to packet reordering and efficiently differentiate packet loss and packet reordering.

2.3.1 TCP selective acknowledgement (SACK) option

To make TCP more robust to multiple losses per congestion window, RFC 2018[19] introduces the TCP Selective Acknowledgement (SACK) option. Upon negotiated by both the sender and the receiver at the start of the TCP connection, a receiver can *"selectively acknowledge"* individual data segments in a congestion window that have been well received. Thus, the sender can selectively retransmit missing data segments. This option requires the sender to maintain a *"scoreboard"* where in each data segment selectively acknowledged by the receiver is *"marked"* as selectively acknowledged. This creates *"holes"* when some data segments have not been received by the receiver. These holes are an indication of either a packet loss or packet reordering. The receiver acknowledges all the data segments in the window with a cumulative acknowledgement once all the individual segments are received.

2.3.2 TCP duplicate selective acknowledgement (DSACK) option

RFC 2883[20] TCP Duplicate Selective Acknowledgement (DSACK) extends the SACK option[19] to enable the receiver to notify the sender about spurious retransmits. While RFC 2018[19] specified the use of the SACK option for acknowledging out-of-sequence data not covered by TCP's cumulative acknowledgement field, RFC 2883[20] suggests the use of SACK[19] to report segment numbers spuriously transmitted by the sender. The

reason for spurious retransmissions can be a result of reordered packets, ACK loss, packet replication and/or early retransmit timeouts. The sender can detect these DSACKs to adapt accordingly with the use of appropriate reaction algorithms.

The use of DSACK does not require separate negotiation between a TCP sender and receiver that have already negotiated SACK capability. The absence of separate negotiation for D-SACK means that the TCP receiver could send DSACK blocks when the TCP sender does not understand this extension to SACK. In this case, the TCP sender will simply discard any D-SACK blocks, and process the other SACK blocks in the SACK option field as it normally would.[20] Furthermore, the DSACKS for a particular segment are only reported once by the receiver. Therefore, if there are heavy reverse path losses, these DSACKS can get lost leaving behind an uninformed sender.

2.3.3 TCP timestamp option

RFC 1323[18] defines the timestamp option in TCP. The Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option. The Timestamp Echo Reply field (TSecr) is only valid if the ACK bit is set in the TCP header; if it is valid, it echos a timestamp value that was sent by the remote TCP in the TSval field of a Timestamps option.

2.3.4 Deployment trends of TCP options

It is important to know the extent of deployment of the above mentioned TCP extensions that will be used in this thesis work to gauge the effectiveness of the algorithm. In this research work[21], the deployment trends of various TCP extensions are studied. Here, we focus on SACK and Timestamp options deployment trends.

The hosts for the test were chosen from Alexa's[22] top 100000 webservers list. The TCP handshake of distinct hosts were captured and analyzed. Out of the 77854 hosts analyzed, they find that 69334 (89.06%) hosts supported SACK and 64220 (83.77%) hosts supported timestamp option.

2.4 Impact of packet reordering on TCP

TCP is designed to handle all sorts of network behaviours and guarantee an in-order delivery to the application layer[1]. With the rise of packet reordering and especially with large-scale reordering, it has multiple destructive implications on TCP. On one hand TCP is unable to grow the congestion window and make use of the available bandwidth and on the other hand is makes TCP lose its self-clocking property and become bursty. Packet reordering can be classified in to forward path reordering or data reordering and reverse path reordering or ACK reordering. This classification is necessary because of

the very asymmetric nature of Internet routing and the different nature of impact it has on TCP in different directions.

2.4.1 Forward path reordering

In forward path reordering, TCP data segments arrive out-of-order at the receiver. This has many implications which are briefly discussed in the following sections.

Spurious retransmits

Most TCP implementations use an algorithm called Fast Retransmit[8] to try to recover as quickly as possible from losses. This is usually triggered by a series of duplicate acknowledgements from the receiver. Usually, after a TCP sender receives three such duplicate acknowledgements, the data after the byte being acked is considered lost and thus retransmitted. This retransmission is spurious in case of packet reordering and when it arrives at the receiver after more than three data segments.

At this point of retransmission at the sender, the sender also reduces the sending rate to half the previous value. In case of persistent reordering or multiple reordering in a single window, the TCP sender reduces the sending rate multiple times and thus unable to make use of the available resources. Furthermore, with every spurious retransmission, the bandwidth is wasted by sending the same data twice. [23] [12]

Obscured packet loss

When packet reordering and packet loss occur in the same window,it can happen that TCP is unable to see the packet loss until a RTO time-out occurs. Let us consider a sender transmits packets 1 through 6. Let us assume that packet 2 is reordered and arrives after 6; packet 3 is lost. The sender receives 3 duplicate acknowledgements for packet 2 and therefore retransmits packet 2. The sender acknowledgement of packet 2(triggered by the reordered packet 2) soon after retransmission. Since there are no more packets in flight, no duplicate acknowledgements are generated by the receiver for packet 3 which is actually lost. This results in a time-out and triggers a loss recovery.

Poor round-trip time calculation

Most TCP implementations calculate round trip time (RTT) either through TCP timestamps or through recording the time taken for a particular sequence number in a window of data to be acknowledged. In case timestamps are enabled in the system and if they are used, the RTT is calculated by the difference in the timestamps echoed in the acks to the current time to get an accurate RTT. In case timestamps are not used, a timer based approach is used. A timer is started for a sequence number when it is transmitted and the time is stopped when an ACK is received for that particular segment number.

However, RTT calculations in these implementations are only valid iff the segment is not

retransmitted. in case of retransmissions including spurious retransmissions, the timer samples are discarded. Since packet reordering causes spurious retransmissions, most of the potential samples are discarded leading to a poor RTT estimation.

TCP receiver efficiency

Packet reordering impacts the operation of the receiver and the receiver application. The TCP receiver has to first buffer all the out of order data so that it can deliver it in-order to the receiving application. At some point in time, the receiver has to spend CPU cycles to order its out-of-order receiver queue. Secondly, the receiving application receives bursty data which results in the decrease of overall efficiency of the system.

Furthermore, TCP implementations use header prediction to reduce the costs on TCP processing. However, it works only on in-order data. So, if segments are reordered, far more processing is required at the receiver side.

2.4.2 Reverse path reordering

In reverse-path reordering, the acks travelling back to the sender are reordered. In reverse-path reordering, the receiver is sending cumulative acks (because data is arriving in order), but the acks are being reordered on their way back to the sender.[12]

The acks are a rough sign of the data rate in the network and how fast the receiver is able to consume the sent data.[24] The major effect of reverse path reordering is the loss of self clocking property of TCP.

Loss of self clocking

The TCP sender receives a stream of acks acknowledging one or two data segments notifying the sender to send more data per acknowledgement. However, with reverse path reordering, the TCP sender sees an ACK acknowledging a large amount of data followed by a stream of acks acknowledging already acknowledged data. This has a direct effect on how the TCP sender opens the congestion window based on whether it is in the slow start phase or congestion avoidance phase.[8] With a reordered ACK acknowledging more data, the following stream of acks are simply discarded by the sender thus reducing the actual number of acks to the number of reordered acks. Since the working of slow start and congestion avoidance heavily depend on acks acknowledging new data, TCP is affected.

The other important effect of reordering in the reverse path is making the TCP sender bursty. with a reordered ACK acknowledging more data segments, more "free" slots are opened. This makes the TCP sender bursty irrespective of if it is in slow start or congestion avoidance. This increases the resource utilization at the receiver and receiver

application. This effect is reinforcing; with every burst, more acks are generated resulting in more ACK reordering due to reverse path reordering. This effect aggravates if ACK compression techniques are used.[25]

2.4.3 Forward and reverse path reordering

It is hard to expect that a network has only forward path or only reverse path reordering. In the presence of reordering in both directions, we can expect the TCP sender to exhibit a stronger forward path or stronger reverse path reordering behaviors or an oscillation between both the behaviors.

2.5 Impact on future technologies

Some of the main areas where a reordering robust TCP would bring in changes to the way devices are deigned and operate are high speed switches [26], multi-path routing, high-delay satellite links. The other areas include Novel routing algorithms, network components, link-layer retransmission mechanisms. A reordering robust TCP can make way for many numerous design possibilities of next gen network components.

2.6 Approaches to make TCP robust to packet reordering

In fig 2.2, the state diagram of a TCP socket handling congestion control and packet loss has been illustrated as specified in [27]. As long as a TCP receiver is sending data segments in-order, the TCP sender remains in OPEN state. Upon the receipt of the first data segment that is out of order at the receiver, the receiver sends an immediate duplicate ACK. The purpose of this ACK is to inform the sender that the segment received was out of order. At the sender-side, the sender is unclear if the data was lost in the network due to network congestion or packet reordering or packet duplication in the network. The sender enters a DISORDER state upon the receipt of the first duplicate ACK. The receiver sends an immediate ACK when the incoming data segment from the sender fills the hole in the receiver sequence space. In such an event, the sender returns to OPEN state.

In the event of a packet loss, TCP sender reduces the CWND, SSTHRESH and uses Fast Retransmit to retransmit the lost packet. At the moment, the trigger from DISORDER to RECOVERY is set at 3 duplicate ACKs. After this retransmission, the Fast Recovery algorithm kicks in. New data segments are transmitted until the sender receives a non duplicate ACK. At this moment, the TCP state changes from RECOVERY to OPEN.

Each time a sender transmit new data segment, each data segment is put in the retransmission queue and a timer is set to each data segment. If an ACK is not received by the sender before the timeout, the TCP sender transits to LOSS state which serves

as a means to restart ACK clock. A possible case of RTO time out can be heavy reverse path losses.

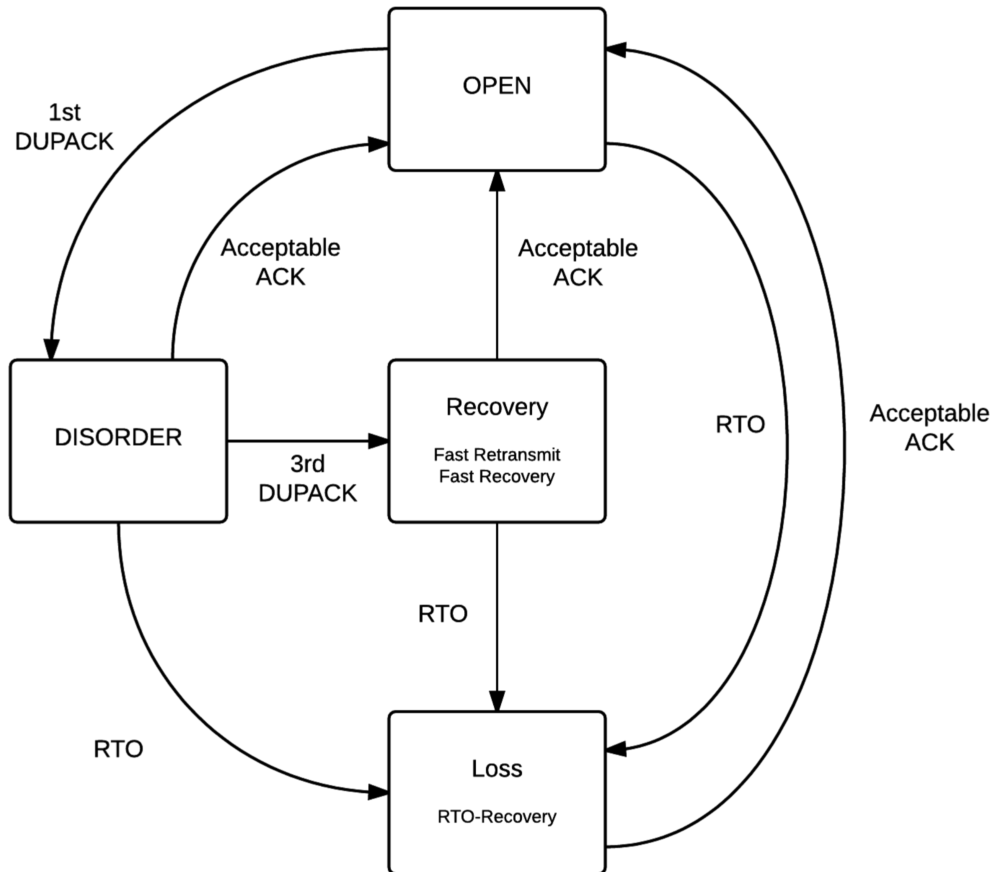


Figure 2.2: TCP congestion control and loss recovery

There are various approaches proposed to precisely differentiate packet loss and packet reordering. The can be classified based on their approach as Proactive, Reactive and tolerant algorithms. Proactive algorithms focus on preventing spurious transitions from DISORDER to RECOVERY state. Reactive algorithms, focus on undoing false transitions when in RECOVERY. Tolerant algorithms, prevent future false transition of state from occurring.

Practices for TCP Senders in the Face of Segment Reordering[28] suggests various proactive ways of preventing spurious transition to RECOVERY. [29] focusses on problems challenges and solutions while tackling packet reordering. [30, 31] are few proactive algorithms that aim at reducing spurious transitions to RECOVERY state. [32, 33] are few other proactive based algorithms using timers. [20, 34, 35, 9] suggest reactive approaches

to undo false transitions to RECOVERY.

This master thesis concentrates on TCP-adaptive Non-Congestion to Robustness(TCP-aNCR)[3, 4], TCP-Non Congestion to Robustness(TCP-NCR)[2] and Linux's implementation of reordering detection and reaction. Linux follows a proactive and tolerant approach to resolving spurious transitions. When Linux detects packet reordering, it sets the DUPTHRESH to maximum reordering extent or 127 (which ever is smaller) thereby delaying the transition to RECOVERY. At the beginning of the connection, the DUPTHRESH is set to 3. Therefore, it falls under proactive and tolerant category. Furthermore, Linux uses timestamps (when enabled) to detect and react to false transitions which is possible if the reordering extent is greater than 127. TCP-NCR falls under proactive category where DUPTHRESH is set to CWND irrespective of the presence or absence of packet reordering. On the other hand, TCP-aNCR follows a proactive, reactive and tolerant approach where at the start of the connection, the DUPTHRESH is set to 3 and upon detecting packet reordering, it avoid future possible incidents of packet reordering by calculating DUPTHRESH based on the reordering extent. It also makes use of timestamps (when available) to react to false transitions which his very useful in slow start phase of the TCP connection when DUPTHRESH is still at 3.

This thesis focusses on the performance of Linux, TCP-NCR and TCP-aNCR under various reordering conditions. TCP-NCR and TCP-aNCR are discussed in detail in the next chapter.

3 TCP-NCR and TCP-aNCR

In this section we discuss in detail TCP Non Congestion to Robustness (NCR) [2] and TCP-adaptive NCR (aNCR) as these algorithms are the primary focus of the thesis.

It is to be noted that both these algorithms are used only when Nagle algorithm [36] is deployed as in its absence, there is no way to accurately calculate the number of outstanding segments in the network (and, therefore, no good way to derive an appropriate duplicate ACK threshold) without adding state to the TCP sender.[2][36] Furthermore TCP SACK[19] must be enabled to indicate the sender the data segments that have been received well by the receiver so that the sender maintains an accurate scoreboard.

3.1 Extended limited transmit (ELT)

Both TCP-NCR and TCP-aNCR increase the duplicate threshold value in the event of packet reordering from the default value of three to say an arbitrary value "X". With a duplicate threshold value of "X", the sender waits for "X" duplicate acknowledgements before it retransmits. During this waiting process, we may run into multiple scenarios.

- The wait for "X" duplicate acks cause the medium to be idle and renders it underutilized
- In the presence of reverse path losses, a couple of acks might be lost thus we will never be able to get "X" duplicate acks. This leaves us in a dangling situation and eventually a RTO timeout.

To prevent this, we extend the limited transmit mechanism[37] to send additional data while the TCP sender is in the process of disambiguating loss vs packet reordering. In [2] the authors suggest two variants of extended limited transmit namely Careful limited transmit (CF) and Aggressive limited transmit (AG).

3.1.1 Careful limited transmit (CF ELT)

In Careful extended limited transmit, a new previously unsent data segment is transmitted for every other duplicate ACK that selectively acknowledges new data. This essentially reduces the sending rate (halves in the case of NCR as we demonstrate later) during the disorder phase when disambiguating packet reordering versus packet loss. If a packet was found to be reordered , TCP returns to the OPEN state. If a packet is lost, the sender proceeds to fast retransmit the lost packet and enter loss RECOVERY state.

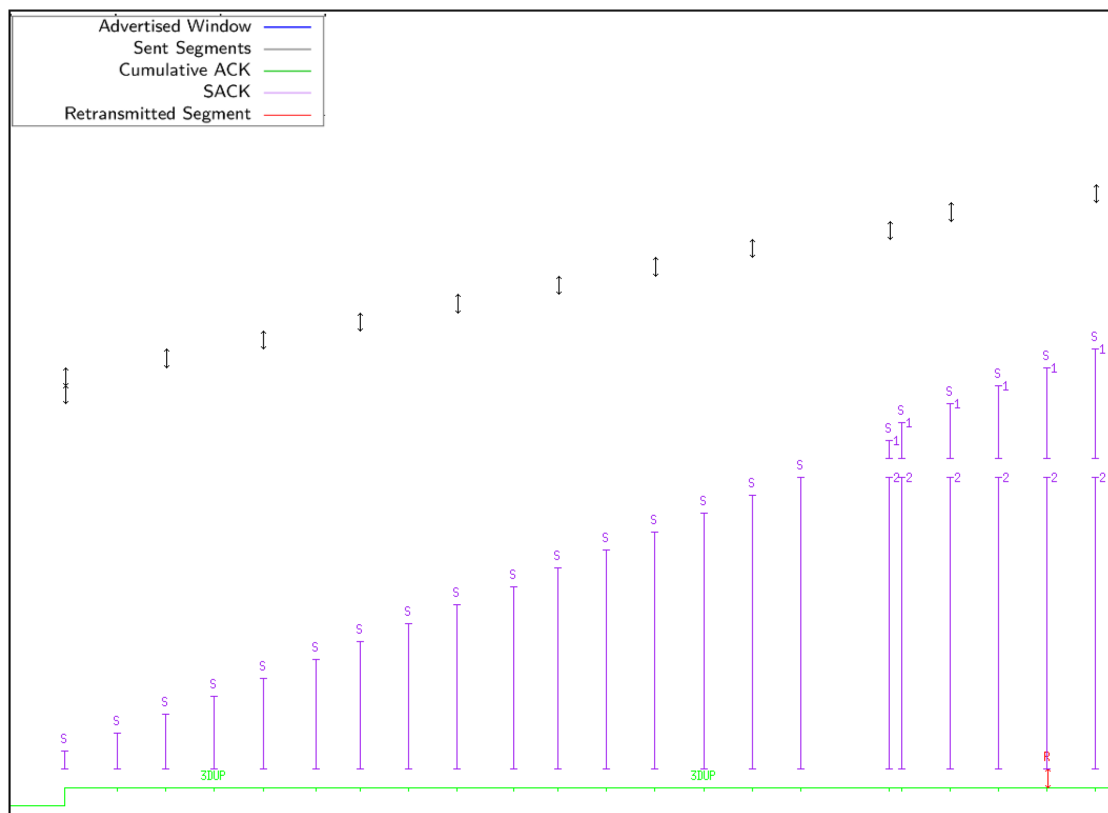


Figure 3.1: Careful limited transmit

If the duplicate threshold is increased from three to congestion window worth of data segments (the case of NCR), then, when this duplicate threshold value is reached, number of out standing data cumulatively acknowledged is exactly 3/2 times the congestion window value. Furthermore, at this point, the sending rate will have reduced to half as we would be sending out new data for every second acknowledgement. In figure 3.1 we can see that when the packet is determined to be in fact, lost; the sending rate has already been reduced in half. The decrease in the width between the new segment transmitted and the highest data segment selectively acknowledged decreases is an indicator of the decreasing sending rate.

3.1.2 Aggresive limited transmit (AG ELT)

In Aggressive extended limited transmit, a new previously unsent data segment is transmitted for every duplicate ACK that acknowledges new data. This essentially maintains the sending rate during the disorder phase when disambiguating packet reordering versus packet loss. If a packet was reordered, TCP returns to the previous state. If a packet is lost, the sender proceeds to fast retransmit and loss RECOVERY state.

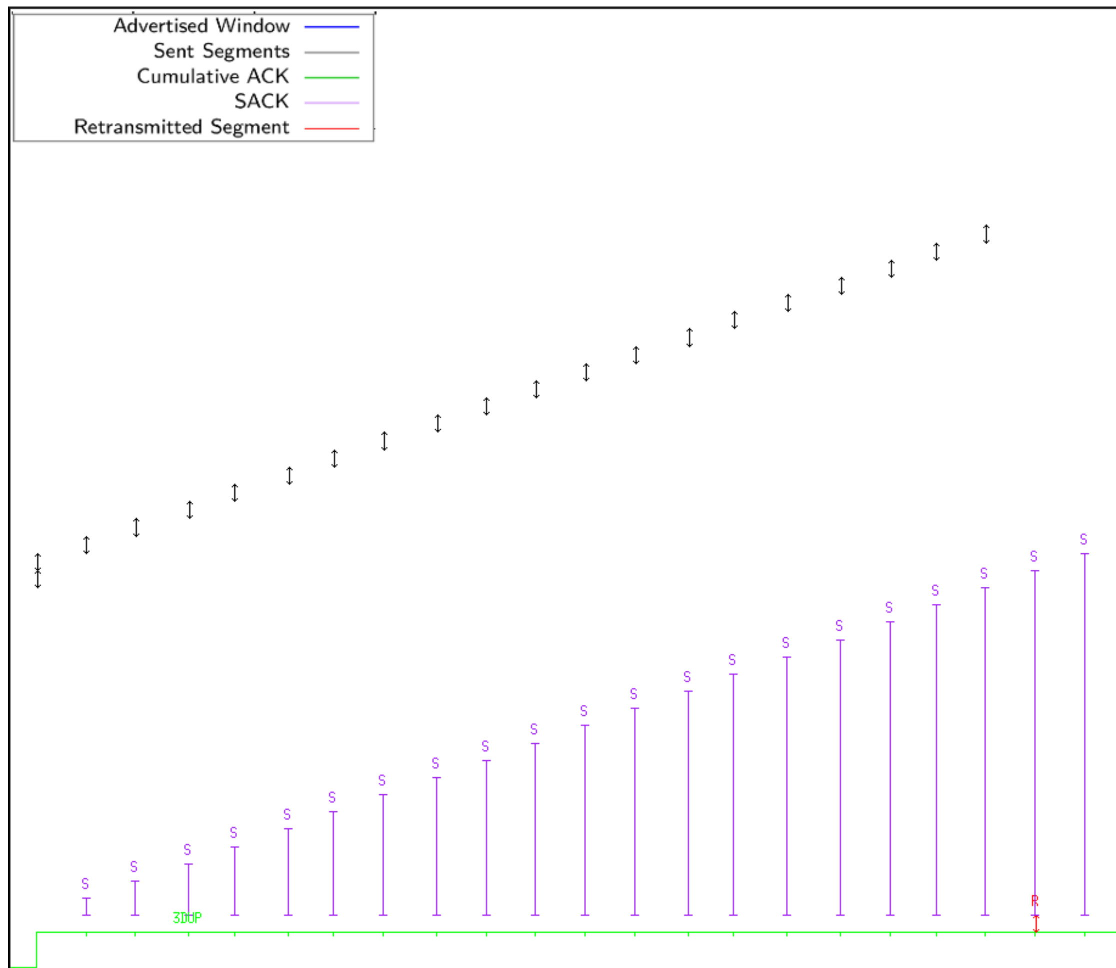


Figure 3.2: Aggresive limited transmit

If the duplicate threshold is increased from three to congestion window worth of data segments, then, when this duplicate threshold value is reached, number of out standing data cumulatively acknowledged is exactly twice the congestion window. In figure 3.2 we can see that when the packet is determined to be in fact, lost; the sending rate has not yet been reduced in half. The constant width between the new segment transmitted and the highest data segment selectively acknowledged is an indicator of the constant sending rate. At this point, when fast retransmit and recovery happens, algorithms such as rate halving or PRR[38] to reduce the sending rate.

3.1.3 Hybrid limited transmit (HY ELT)

While careful limited transmit reduces the sending rate to almost half of its original value at the end of 1 RTT in DISORDER phase, Aggressive limited transmit, main-

tains the same sending rate in DISORDER for almost 1 RTT, if a packet is indeed lost, the congestion control algorithms have to take care of reduction or pacing of sending rate. many congestion control algorithms employ various algorithms to determine the sending rate in RECOVERY. This thesis proposes "Hybrid limited transmit" a supplement to "careful limited transmit" where, the target sending rate in RECOVERY is pre-emptively calculated for the corresponding congestion control algorithm. Many congestion algorithms such as [39, 40] do not exactly reduce the sending rate to half in RECOVERY. Hybrid ELT is compatible with these algorithms as it switches from careful limited transmit to aggresive limited transmit in ELT when the target sending rate is reached.

3.2 TCP-NCR

TCP-NCR changes the trigger for retransmitting a segment is changed from three duplicate ACKs[41] [42] to indications that a congestion window's worth of data has left the network.

Furthermore, TCP-NCR decouples initial congestion control decisions from retransmission decisions, in some cases delaying congestion control changes relative to TCP's current behavior as defined in [41]. The algorithm also suggests the use of either CF ELT or AG ELT. Depending on which type of ELT is used, a constant LT_F is defined as the inverse of the maximum flight size at the end of the duplicate threshold value. Therefore,

- Careful limited transmit: $LT_F = 2/3$
- Aggresive limited transmit: $LT_F = 1/2$

This constant is defined to ensure that enough number of packets are SACKed and that, the packet is truly lost.

The algorithm has been described below as defined in rfc 4653[2] for the convenience of explaining the highlights and pitfalls of the algorithm.

3.2.1 Algorithm

Initialization

For every first sack block received at the sender,

```
(I.1) The TCP MUST save the current FlightSize.

      FlightSizePrev = FlightSize

(I.2) The TCP MUST set a variable for tracking the number of
```

```
        segments for which an ACK does not trigger a transmission
        during Careful Limited Transmit.

        Skipped = 0

        (Note: Skipped is not used during Aggressive Limited
        Transmit.)

  (I.3) The TCP MUST set DupThresh (from [RFC3517]) based on the
        current FlightSize.

        DupThresh = max (LT_F * (FlightSize / SMSS),3)
```

Note: The lower bound of DupThresh = 3 from [RFC2581, RFC3517][41, 42] is kept. In addition to the above steps, the incoming ACK MUST be processed with the steps in section 3.2.1.

ELT termination

The arrival of an ACK that advances the cumulative ACK point while in Extended Limited Transmit, but before loss recovery is triggered, signals that a series of duplicate ACKs was caused by reordering and not congestion. Therefore, the receipt of an ACK that extends the cumulative ACK point MUST terminate Extended Limited Transmit. As described below (in (T.4)), an ACK that extends the cumulative ACK point and *also* contains SACK information will also trigger the beginning of a new Extended Limited Transmit phase.

Upon the termination of Extended Limited Transmit, and especially when using the Careful variant, TCP-NCR may be in a situation where the entire cwnd is not being utilized, and therefore TCP-NCR will be prone to transmitting a burst of segments into the network. Therefore, to mitigate this bursting when a TCP-NCR in the Extended Limited Transmit phase receives an ACK that updates the cumulative ACK point (regardless of whether the ACK contains SACK information), the following steps MUST be taken:

```
  (T.1) A TCP MUST reset cwnd to:

        cwnd = min (FlightSize + SMSS,FlightSizePrev)

        This step ensures that cwnd is not grossly larger than the
        amount of data outstanding, a situation that would cause a
        line rate burst.

  (T.2) A TCP MUST set ssthresh to:
```

```
        ssthresh = FlightSizePrev

        This step provides TCP-NCR with a sense of "history".  If step
        (T.1) reduces cwnd below FlightSizePrev, this step ensures that
        TCP-NCR will slow start back to the operating point in effect
        before Extended Limited Transmit.

   (T.3) A TCP is now permitted to transmit previously unsent data as
        allowed by cwnd, FlightSize, application data availability, and
        the receiver's advertised window.

   (T.4) When an incoming ACK extends the cumulative ACK point and also
        contains SACK information, the initializations in steps (I.2)
        and (I.3) MUST be taken (but step (I.1) MUST
        NOT be executed) to re-start Extended Limited Transmit. In
        addition, the "E" series of steps MUST be taken.
```

During extended limited transmit

Once NCR has already entered the extended limit transmit phase, the following steps are taken to have the ACK clock ticking as described in section 3.1

```
   (E.1) The SetPipe () procedure from [RFC3517] MUST be used to set
        the "pipe" variable (which represents the number of bytes
        still considered "in the network").  Note: the current value
        of DupThresh MUST be used by SetPipe () to produce an accurate
        assessment of the amount of data still considered in the
        network.

   (E.2) If the comparison in equation (1), below, holds and there are
        SMSS bytes of previously unsent data available for
        transmission, then the sender MUST transmit one segment of SMSS
        bytes.

          (pipe + Skipped) <= (FlightSizePrev - SMSS)              (1)

        If the comparison in equation (1) does not hold or no new data
        can be transmitted (due to lack of data from the application
        or the advertised window limit), skip to step (E.6).

   (E.3) Pipe MUST be incremented by SMSS bytes.
```

```
(E.4) If using Careful Limited Transmit, Skipped MUST be incremented
      by SMSS bytes to ensure that the next SMSS bytes of SACKed data
      processed does not trigger a Limited Transmit transmission
      (since the goal of Careful Limited Transmit is to send upon
      receipt of every second duplicate ACK).

(E.5) A TCP MUST return to step (E.2) to ensure that as many bytes
      as are appropriate are transmitted.  This provides robustness
      to ACK loss that can be (largely) compensated for using SACK
      information.

(E.6) DupThresh MUST be reset via:

        DupThresh = max (LT_F * (FlightSize / SMSS),3)

      where FlightSize is the total number of bytes that have not
      been cumulatively acknowledged (which is different from
      "pipe").
```

Entering loss recovery

Once the "DUPTHRESH" number of DUPACKs have been received while in ELT phase, we should enter loss recovery. The ssthresh and DUPTHRESH values are to be set based on the value of FlightSizePrev to reflect the state prior to entering loss recovery.

`ssthresh = cwnd = (FlightSizePrev / 2)`

3.2.2 Drawbacks of TCP-NCR

There are some critical flaws in the NCR algorithm which limits the performance of TCP in the presence of packet reordering. Firstly, in the step T.2 3.2.1 the slow start threshold value is reset to the flight size value prior to entering ELT (FlightSizePrev). If ELT is entered for the first time, the ssthresh is infinity. Thus, this prevents the connection to resume slow start in case of packet reordering during slow start. This has serious implications on the performance of a TCP connection.

Secondly, each time a packet is really lost and not reordered, it takes a RTT to realize and enter recovery state. This increases the application perceived latency in the network which would lead to a huge impact in applications such as high frequency trading, real time bidding, caching databases, autonomous decision systems.

Thirdly, in the presence of continuous packet reordering, the TCP socket is always in extended limit transmit phase (see step T.4) which leads to decreased(in case of CF ELT) / constant(in case of AG ELT) transmission rate as the congestion window never grows.

3.3 TCP-adaptive NCR

TCP adaptive NCR addresses the main pitfalls of NCR and furthermore, decouples the logic for packet reordering detection and packet reordering reaction. In the following subsections, the steps for packet reordering detection and packet reordering reaction are discussed in detail.

3.3.1 Reordering detection

TCP adaptive NCR reordering detection algorithm makes use of TCP timestamps, and SACK and DSACK to accurately detect packet reordering in the network. The algorithm provides a set of metric that can be used by any reordering reaction algorithm to handle packet reordering. There are a few pre requisites to be taken care when implementing the aNCR reordering detection algorithm. Care should be taken to ensure that no other loss recovery algorithm modifies the TCP control block and the SACK scoreboard.

The algorithm has been detailed below as in the internet draft [3]

Initialization

Upon the establishment of a TCP connection, the following variables must be initialized in the TCP control block.

```
(C.1)  The variable Dsack, which indicates whether a DSACK has been
       received so far, and the data structure Samples, which stores
       the computed reordering extents, MUST be initialized as:

          Dsack = false
          Samples = []

(C.2)  If the TCP Timestamps option [RFC1122] has been negotiated,
       then the variable Timestamps MUST be activated and the data
       structure Retrans_TS, which stores the value of the TSval
       field of the retransmissions sent during Fast Recovery, MUST
       be initialized.  Additionally, the data structure
       Retrans_Dsack MAY be used in order to detect reordering longer
       than RTT with Timestamps and DSACK:

          Timestamps = true
          Retrans_TS = []
          Retrans_Dsack = []

       Otherwise, the Timestamps-based detection SHOULD be
       deactivated:
```

```
      Timestamps = false
```

Receiving acknowledgments

```
For each received ACK that either a) carries SACK information, *or*
b) is a full ACK that terminates the current Fast Recovery procedure,
*or* c) is an acceptable ACK that is received immediately after a
duplicate ACK, execute steps (A.1) to (A.4), otherwise skip to step
(A.4).

(A.1)  If a) the ACK carries new SACK information, *and* b) the SACK
       scoreboard is empty

          FlightSizePrev = FlightSize


(A.2)  If the received ACK either a) cumulatively acknowledges at
       most SMSS bytes, *or* b) selectively acknowledges at most SMSS
       bytes in the sequence number space in the SACK scoreboard,
       then:

          The TCP sender MUST execute steps (S.1) to (S.4)

(A.3)  If a) Timestamps == false *and* b) the received ACK carries a
       DSACK option [RFC2883] and the segment identified by the DSACK
       option can be marked according to step (A.1) to (A.4) of
       [RFC3708] as a valid duplicate, then:

          The TCP sender MUST execute steps (D.1) to (D.3)

(A.4)  The TCP sender MUST terminate the processing of the ACK by
       this algorithm and MUST continue with the default processing
       of the ACK.
```

Receiving acknowledgments closing hole

```
(S.1)  If (a) the newly cumulatively or selectively acknowledged
       segment SEG is a retransmission *and* b) both equations Dsack
       == false and Timestamps == false hold, then the TCP sender
       MUST skip to step (A.4).

(S.2)  Compute the relative and absolute reordering extent ReorExtR,
       ReorExtA:
```

```
      The TCP sender MUST execute steps (E.1) to (E.4)

(S.3) If a) the newly acknowledged segment SEG was not retransmitted
      before *or* b) both equations Timestamps == true and
      Retrans_TS[SEG.SEQ] > ACK.TSecr hold, i.e., the ACK
      acknowledges the original transmission and not a
      retransmission, then hand over the reordering extents to an
      additional reaction algorithm.

(S.4) If a) the previous step (S.3) was not executed *and* b) both
      equations Dsack == true and Timestamps == false hold, save the
      reordering extents for the newly acknowledged segment SEG for
      at least two RTTs:

         Samples[SEG.SEQ].ReorExtR = ReorExtR
         Samples[SEG.SEQ].ReorExtA = ReorExtA

(S.5) If a) the newly acknowledged segment SEG was retransmitted
      before exactly once *and* b) both equations Dsack == true and
      Timestamps == true hold *and* c) Retrans_TS[SEG.SEQ] ==
      ACK.TSecr, then save FlightSizePrev for this segment in order
      to be calculate the metrics in case a DSACK arrives, i.e.
      reordering delay is greater than RTT:

      Retrans_Dsack[SEG.SEQ] = FlightSizePrev
```

Receiving duplicate selective acknowledgement

```
(D.1) If no DSACK has been received so far, the sender MUST set:

         Dsack = true

(D.2) If a) the previous step (D.1) was not executed *and* a
      reordering extent was calculated for the segment SEG
      identified by the DSACK option, then the TCP sender MUST
      restore the values of the variables ReorExtR and ReorExtA and
      delete the corresponding entries in the data structure:

         ReorExtR = Samples[SEG.SEQ].ReorExtR
         ReorExtA = SAMPLES[SEG.SEQ].ReorExtA

(D.3) If a) step (D.1) was not executed *and* b) FlightSizePrev was
```

```
        saved in step (S.4) for the segment, then the TCP sender MUST
        calculate the reordering extent for the segment with the E
        series of steps by using the FlightSizePrev saved for this
        segment and afterwards delete the corresponding entries:

           FlightSizePrev_saved = Retrans_Dsack[SEG.SEQ]

(D.4)   Hand the new reordering extents over to an additional reaction
        algorithm.
```

Computing reordering extent

```
(E.1)   SEG.SEQ is the sequence number of the newly cumulatively or
        selectively acknowledged segment SEG.

(E.2)   SND.FACK is the highest either cumulatively or selectively
        acknowledged sequence number so far plus one.

(E.3)   The TCP sender MUST compute the absolute reordering extent
        ReorExtA as

        ReorExtA = (SND.FACK - SEG.SEQ) / SMSS

(E.4)   The TCP sender MUST compute the relative reordering extent
        ReorExtR as

        ReorExtR= ReorExtA * (SMSS / FlightSizePrev)
```

Retransmitting segment

```
If the TCP Timestamps option [RFC1323] is used to detect packet
reordering, the TCP sender must save the TCP Timestamps option of all
retransmitted segments during Fast Recovery.

(RET)   If a) a segment SEG is retransmitted during Fast Recovery,
        *and* b) the equation Timestamps = true holds, the TCP sender
        MUST save the value of the TSval field of the retransmitted
        segment:

        Retrans_TS[SEG.SEQ] = SEG.TSval
```

Retransmission timeout

In the event of a RTO timeout, it is an indication that the values of reordering extents are outdated due to change in path characteristics and therefore, the data structures need to be reset.

```
Samples = []
Retrans_TS = []
FlightSizePrev = 0
```

3.3.2 Reordering reaction

TCP aNCR reordering reaction algorithm makes use of the reordering metrics provided by aNCR reordering detection algorithm and handles events of packet reordering and packet loss. It is capable of dynamically changing the DUPTHRESH value based on the value of relative reordering extent ReorExtR provided by aNCR reordering detection algorithm or have a constant DUPTHRESH as in TCP NCR. The benefits of having a dynamic DUPTHRESH is latency benefits. However, the amount of latency benefits needs to be analysed.

The following sections describe the aNCR reordering reaction algorithm as specified in the internet draft [4].

Initialization

```
Depending on the type of extended limit transmit used, we have to set the
"LT_F" constant as:

   if CF ELT:
    LT_F = 2/3

   elif AG ELT:
    LT_F = 1/2

Depending on the type of DUPTHRESH preferred, dynamic or constant value:

   if dynamic:
    ReorExtR = 0
   else:
    ReorExtR = -1

Upon receiving a DUPACK, if the SACK scoreboard is found to be empty, it is a sign of the st
phase to determine if the packet is lost or reordered. ELT needs to be initialized upon the
DUPACK with a SACK block selectively acknowledging new data.
```

```
The following steps should be performed.

   (I.1)  The TCP sender MUST save the current outstanding data:

             FlightSizePrev = FlightSize

   (I.2)  The TCP sender MUST save the highest sequence number
          transmitted so far:

             recover = SND.NXT - 1

   (I.3)  The TCP sender MUST initialize the variable 'skipped' that
          tracks the number of segments for which an ACK does not
          trigger a transmission during Careful Limited Transmit:

             skipped = 0

          During Aggressive Limited Transmit, 'skipped' is not used.

   (I.4)  The TCP sender MUST set DupThresh based on the current
          FlightSize:

             DupThresh = max (LT_F * (FlightSize / SMSS), 3)

          The lower bound of DupThresh = 3 is kept from [RFC5681]
          [RFC6675].

   (I.5)  If (ReorExtR != -1) holds, then the TCP sender MUST set
          DupThresh based on the relative reordering extent 'ReorExtR':

             DupThresh =
                  max (min (DupThresh,
                            ReorExtR * (FlightSizePrev / SMSS)), 3)
```

As described in step I.5, if dynamic DUPTHRESH is chosen for the connection, the DUPTHRESH is calculated based on the relative reordering extent (ReorExtR) provided by the aNCR packet reordering detection algorithm. In the absence of packet reordering, the DUPTHRESH value is 3 and in the worst case scenario, the DUPTHRESH value is the same as NCR.

If a path exhibits constant moderate rate of packet reordering, with aNCR reaction algorithm, one should see lesser application perceived latency compared to NCR. How-

ever, the characteristics of packet reordering in the wild is something to be researched upon.

ELT termination

Upon the arrival of a DUPACK after entering the ELT phase and iff the DUPACK advances the cumulative ACK point, it is a sign of packet reordering. The following steps need to be performed.

```
(T.1)  If the received ACK extends not only the cumulative ACK point,
       but *also* carries new SACK information , the TCP sender MUST
       restart Extended Limited Transmit and MUST go to step (T.2).
       Otherwise, the TCP sender MUST terminate it and MUST skip to
       step (T.3).

(T.2)  If the Cumulative Acknowledgment field of the received ACK
       covers more than 'recover' (i.e., SEG.ACK > recover), Extended
       Limited Transmit has transmitted one cwnd worth of data
       without any losses and the TCP sender MUST update the
       following state variables by

          FlightSizePrev = pipe_max
          pipe_max = 0

       and MUST go to step (I.2) to re-start Extended Limited
       Transmit.  Otherwise if (SEG.ACK <= recover) holds, the TCP
       sender MUST go to step (I.3).  This ensures that in the event
       of a loss the cwnd reduction is based on a current value of
       FlightSizePrev.

The following steps are executed only if the received ACK does *not*
carry SACK information.  Extended Limited Transmit will be
terminated.

(T.3)  A TCP sender MUST set ssthresh to:

          ssthresh = max (cwnd, ssthresh)

       This step provides TCP-aNCR with a sense of "history".  If the
       next step (T.4) reduces the congestion window, this step
       ensures that TCP-aNCR will slow-start back to the operating
       point that was in effect before Extended Limited Transmit.
```

```
(T.4)  A TCP sender MUST reset cwnd to:

          cwnd = FlightSize + SMSS

       This step ensures that cwnd is not significantly larger than
       the amount of data outstanding, a situation that would cause a
       line rate burst.

(T.5)  A TCP is now permitted to transmit previously unsent data as
       allowed by cwnd, FlightSize, application data availability,
       and the receiver's advertised window.
```

As we can see from the above ELT termination algorithm of aNCR, it addresses the primary disadvantages of NCR. In T.4, the slow start bug of NCR is resolved. If packet reordering occurs during slow start upon returning to OPEN state, ssthresh value is still infinity and therefore, the connection resumes with slow start.

The second major pitfall of NCR is addressed in step T.2, In case of continuous disorder phases, flightsizeprev is updated with the maximum amount of data sent in the last RTT. This ensures that congestion window increases every RTT if it could send all data in the previous RTT without losses.

During extended limited transmit

Once the TCP connection has entered the ELT as described in initialization, if the incoming DUPACK contains a new SACK block and doesn not advance the cumulative ACK point, the following steps are performed.

```
(E.1)  The TCP sender MUST update the SACK scoreboard and uses the
       SetPipe() procedure from [RFC6675] to set the 'pipe' variable.
       Note: the current value of DupThresh MUST be used
       by SetPipe() to produce an accurate assessment of the amount
       of data still considered in the network.

(E.2)  The TCP sender MUST initialize the variable 'burst' that
       tracks the number of segments that can at most be sent per ACK
       to the size of the Initial Window (IW) [RFC5681]:

          burst = IW

(E.3)  If a) (cwnd - pipe - skipped >= 1 * SMSS) holds, *and* b) the
       receive window (rwnd) allows to send SMSS bytes of previously
       unsent data, *and* c) there are SMSS bytes of previously
```

```
        unsent data available for transmission, then the TCP sender
        MUST transmit one segment of SMSS bytes.  Otherwise, the TCP
        sender MUST skip to step (E.7).

(E.4)   The TCP sender MUST increment 'pipe' by SMSS bytes and MUST
        decrement 'burst' by SMSS bytes to reflect the newly
        transmitted segment:

           pipe = pipe + SMSS
           burst = burst - SMSS

(E.5)   If Careful Limited Transmit is used, 'skipped' MUST be
        incremented by SMSS bytes to ensure that the next SMSS bytes
        of SACKed data processed do not trigger a Limited Transmit
        transmission.

           skipped = skipped + SMSS

(E.6)   If (burst > 0) holds, the TCP sender MUST return to step (E.3)
        to ensure that as many bytes as appropriate are transmitted.
        Otherwise, if more than IW bytes were SACKed by a single ACK,
        the TCP sender MUST skip to step (E.7).  The additional amount
        of data becomes available again by the next received duplicate
        ACK and the re-execution of SetPipe().

(E.7)   The TCP sender MUST save the maximum amount of data that is
        considered to have been in the network during the last RTT:

           pipe_max = max (pipe, pipe_max)

(E.8)   The TCP sender MUST set DupThresh based on the current
        FlightSize as described in step I.4.

(E.9)   If (ReorExtR != -1) holds, then the TCP sender MUST set
        DupThresh based on the relative reordering extent 'ReorExtR' as
        described in step I.5.
```

Entering loss recovery

If Proportional rate reduction (PRR) [43] is used, the RecoverFS should be set to FlightSizePrev.

```
RecoverFS = FlightSizePrev
```

If Fast recovery algorithm is used, the ssthresh and DUPTHRESH should be set as described by NCR.

```
ssthresh = cwnd = (FlightSizePrev / 2)
```

Furthermore, each time aNCR reordering detection algorithm providdes new metrics of ReorExtR, the maximum value of is considered.

```
ReorExtR = min (max (ReorExtR, ReorExtR_New), 1)
```

4 Measurement Setup

This chapter describes each and every aspect of our measurement set up used for evaluating the various reordering algorithms. Firstly, the hardware and topology of the network are discussed. From then on, the process of emulating various network conditions is described. In the experiments, the network performance measurement tool Flowgrind is used to perform the experiments. This chapter also explains in detail the implementation of the reordering algorithms in the Linux kernel and reproduction of packet reordering in the testbed.

4.1 Testbed

This thesis focusses the implementation and evaluation of TCP-NCR and TCP-aNCR under two different network topologies. In the first topology (fig: 4.1), there are two senders and two receivers and a number of nodes in between. Each node is responsible for emulating a single unique network characteristics in the network. The network characteristics in question are, forward path delay, forward path reordering, reverse path delay, reverse path reordering, bottleneck and reverse path loss. The purpose of having such a set up is to have a finer control on each of the network characteristics. The network characteristics for the experiments have been adopted as described in Common TCP Evaluation Suite[44]. All the nodes and the nodes are connected by 10GbE.

The purpose of having multiple senders and multiple receivers is to generate cross traffic in the network. One pair of sender-receiver act as a generator and sink sending traffic through the bottleneck node.

All the nodes run vanilla Linux kernel version 3.16 except a pair of sender-receiver which use our modified kernel which is based off vanilla Linux kernel version 3.16 and has TCP-NCR and TCP-aNCR implemented. Furthermore, the reordering nodes run a modified version of the kernel to perform advanced reordering of packets. In the second topology (fig: 4.2), there are no artificial network characteristics introduced except for the forward path reordering. Also in this set up, the nodes act as aggregation switches. The set up still contains two sources and two receivers with a pair acting as a cross traffic generator and sink. Furthermore, at the senders and receivers, a number of sysctl parameters are set. The listing 4.1 specifies all the sysctl changes made prior to the experiments. The buffer autotuning at the receiver is disabled and the buffer size manually set to 30Mb. This has been done because, in our earlier experiments, we came across scenarios where buffer auto tuning did not work in the presence of packet reordering

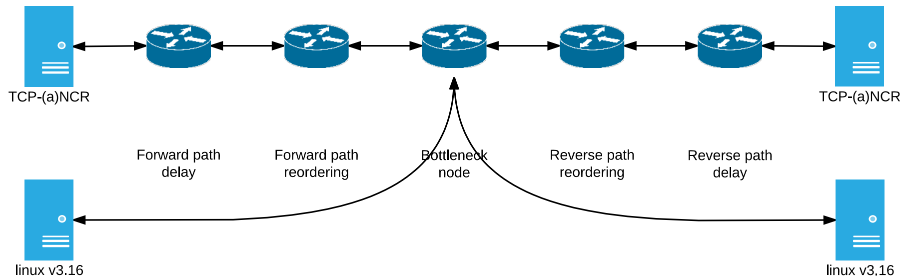


Figure 4.1: Testbed for evaluating under low data rates

thus limiting the sending rate to size of the window set by the faulty auto tuning logic at the receiver.

```
net.ipv4.tcp_no_metrics_save=1
net.ipv4.tcp_congestion_control=reno
net.ipv4.tcp_sack=1
net.ipv4.tcp_dsack=1
net.ipv4.tcp_timestamps=0
net.ipv4.tcp_fack=0
net.ipv4.tcp_ecn=0
net.ipv4.tcp_mtu_probing=0
net.ipv4.tcp_frto=0
net.ipv4.tcp_early_retrans=0
net.ipv4.tcp_moderate_rcvbuf=0
net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_fastopen=0
net.core.rmem_max=67108864
net.core.wmem_max=67108864
net.core.rmem_default=33554432
net.core.wmem_default=33554432
net.ipv4.tcp_rmem="33554432 33554432 67108864"
net.ipv4.tcp_wmem="33554432 33554432 67108864"
```

Listing 4.1: Testbed sysctl settings

The Linux kernel's feature of saving various metrics of previous connections are disabled so that each connection under evaluation is considered as a new connection. SACK and DSACK are enabled. Timestamps are disabled to find the worst case performance of Linux's native algorithm, NCR and aNCR under packet reordering. Various other TCP related kernel parameters are disabled.

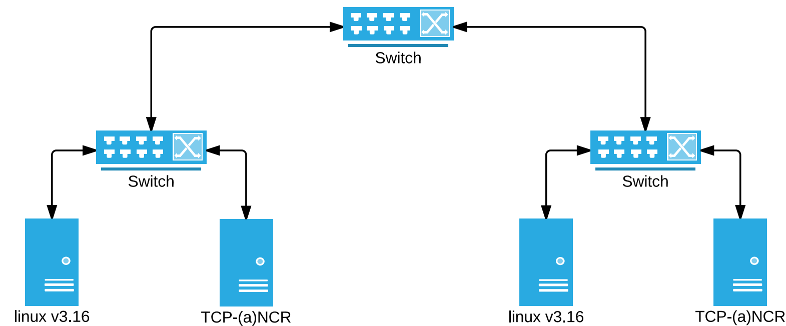


Figure 4.2: Testbed for evaluating under higher data rates

4.2 Traffic shaping and policing

For traffic shaping and policing, Linux's built-in utility Netem[5] is used. The Linux man page[6] is another great resource for detailed list of features regarding traffic policing and shaping using netem. However, since these two references are not maintained well, some of the new features and modified features are not part of the documentation. For a thorough understanding of netem, one needs to analyse the source code of netem in the Linux kernel and also the source code of the userland utility iproute2[45] Firstly, the Linux kernel needs to be configured and netem enabled as explained in listing 4.2

```
Networking --->

    Networking Options --->

      QoS and/or fair queuing --->

          Network emulator
```

Listing 4.2: Configuring Linux kernel for NetEm

For an effective traffic shaping and policing, three critical elements are required. They are a queueing discipline(qdisc), class and a filter[46].

A qdisc is in its primitive form, a packet scheduler. It determines the way we send data out in the network. There are two types of qdiscs; classful based and classless. A classful qdisc can have multiple classes, all of which are internal to the qdisc, and provides

a handle that can be used to attach filters. In classless queueing, a qdisc has no children.

In our set up, we use classful traffic shaping and policing[47]. This is because, this gives us more control on the traffic shaping and policing. For example, in our use case, with classful qdiscs, it is possible to emulate network characteristics such as only forward path delay or only forward path reordering on traffic originating or destined to a particular ip address using filters. We use two child classes in our setup; one for forward path and one for reverse path. The qdiscs used in our experiments is hierarchical token bucket[48] Furthermore, there is no network emulation being set at the senders and the receivers. All network emulation is done on the intermediate routers.

```
tc qdisc add dev eth0 root handle 1: htb default 100

tc class add dev eth0 parent 1: classid 1:1 htb rate 100mbit
```

Listing 4.3: Setting up basic HTB bucket on a node

In the listing 4.3, we are attaching a root class to the ethernet device "eth0" and create a child class with the handle 1:1 which implements a hierarchical token bucket(htb). Now, a filter has to be added to route all the traffic to a particular destination into this qdisc. All the other traffic that do not match the filter are routed through the default bucket 100.

```
tc filter add dev eth0 parent 1: protocol ip prio 1 u32 \
flowid 1:1 match ip dst 192.168.2.219
```

Listing 4.4: Setting up a filter for the HTB bucket

From the listing 4.4, we set a filter at root class to route all traffic destined to the ip address to the bucket 1:1. Now, we have set up a class based queueing where in all packets destined to a particular ip address are routed to our child class at 1:1. However, for example, if we want to treat these packets differently by delaying each packet by 25ms and a jitter of 5ms, we have to set up a network emulation on this particular bucket.

```
tc qdisc add dev eth0 parent 1:1 handle 10: \
rate 1000 mbit netem delay 25ms 5ms
```

Listing 4.5: Setting up a delay for the HTB bucket

With the above command 4.5 we set up our desired delay. However, since jitter causes packet reordering in itself and this is undesired in this specific node, we need to set a high rate on this child bucket. This feature has been hugely debated on the netem mailing list[49].

4.3 Packet reordering in Linux kernel

The Netem implementation in the Linux kernel provides the users with two primary ways of generating packet reordering. The first method allows the users to have packet

reordering every n^{th} packet. As described in the listing 4.6 this example command reorders every 5th packet. The implementation of this feature is so that, the 5th packet is sent immediately and the first 4 packets are sent with a delay of 10ms. There are two drawbacks of using this feature; packet reordering is deterministic and instead of the packet eligible for reordering being reordered, the packets prior to this packet is reordered.

```
tc qdisc change dev eth0 root netem gap 5 delay 10ms
```

Listing 4.6: Packet reordering with gap

The second method of implementing packet reordering using netem is as shown in listing 4.7. In this example, 25% of packets (with a correlation of 50%) will get sent immediately, others will be delayed by 10ms. However, even this method is unacceptable because instead of reordering a single packet, packets prior to this packet are reordered. Thus the Linux kernel's behaviour was changed to reflect our desired behaviour.

```
tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
```

Listing 4.7: Packet reordering with correlation

In the modified version of the Linux kernel, when packet reordering is set using the command in listing 4.7 the packet which is determined to be reordered based on the reordering rate (25%) and correlation (50%) is reordered by 10ms.

4.4 Pluggable framework for reordering algorithms

The vanilla version of the Linux kernel is modified to introduce a pluggable reordering algorithm framework. This enables us to use any reordering algorithms while opening a socket. The algorithms can be chosen via the socket option. Thus, a specific reordering algorithm can be specified via flowgrind itself and the experiments are performed with the chosen algorithm.

This immensely fastens the experiment process and avoids the need of maintaining multiple Linux kernels with multiple reordering algorithms. The pluggable framework was implemented in Linux kernel version 3.16. In the absence of mentioning any reordering algorithm, the kernel falls back to the Linux native behaviour of handling packet reordering.

The other important advantage of having a pluggable framework in the Linux kernel is that, it enables us to try new variants of the reordering algorithms and enables us to test them using our automated scripts against native Linux behaviour, NCR and aNCR to see if these new changes bring any new improvements to the performance.

4.5 Flowgrind: Network performance tool

Flowgrind[7] is a TCP traffic generator tool used for benchmarking TCP/IP stacks. It is compatible with OSX, Linux and FreeBSD. It employs a distributed architecture with daemons running on all the machines whose performance is under test. The controller is responsible for accepting user inputs and passing it on to the daemons to start the experiment. The controller can be anywhere in the network and not necessarily on the machines under test.

The reason for choosing flowgrind over other network performance measurement tools is, the experimental logs are very verbose and in addition to throughput also measures the application layer interarrival time (IAT) and round-trip time (RTT), blockcount and network transactions/s. Unlike most cross-platform testing tools, flowgrind can output some transport layer information, which are usually internal to the TCP/IP stack. For example, on Linux 4.3 this includes among others the kernel's estimation of the end-to-end RTT, the size of the TCP congestion window (CWND), slow start threshold (SSTHRESH)[7], the number of SACKS, DSACKS, retransmissions, spurious retransmissions. These values can be used to get an overview of the TCP connection without actually getting generating xplots from the TCP dumps every single time.

Furthermore, flowgrind also has the option of capturing TCP traces and also specifying the socket options such as congestion control algorithm to be used for that particular connection. Flowgrind was extended to make use of the pluggable reordering kernel where we implemented the option of choosing reordering algorithms via a socket option.

Through out our experiments, we measure and evaluate the performance of TCP NCR and TCP aNCR under bulk traffic as well as request-response traffic. By default, in Flowgrind, bulk transfer tests are performed. However, we can also describe the traffic characteristics which enables us to have our own traffic matrix. We make use of this feature from Flowgrind to describe our request-response traffic as mentioned in cdma2000 Evaluation Methodology. [50]

```
ID  begin    end        through transac min RTT avg RTT max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]       [Mbit/s]   [#/s]    [ms]    [ms]    [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.000 0.050 1150.577258     0.00     inf     inf     inf     inf     inf     inf   10 INT_MAX   10    0    0    0    0    0    3    0  43.5   21.8  244.0     open  1448  1500    0     0     0     3     0     0
 0 0.000 0.050    0.000000     0.00     inf     inf     inf     inf     inf     inf    0       0    0    0    0    0    0    0    0    0   0.0    0.0    0.0     open     0     0    0     0     0     0     0     0
 0 0.050 0.106  987.018000    17.93  42.764  42.764  42.764     inf     inf     inf   20 INT_MAX   20    0    0    0    0    0    3    0  42.7    2.6  244.0     open  1448  1500    0     0     0     3     0     0
 0 0.050 0.102    0.006241     0.00     inf     inf     inf     inf     inf     inf   10 INT_MAX    1    0    0    0    0    0    3    0  40.0   20.0  240.0     open  1448  1500    0     0     0     3     0     0
ID  begin    end        through transac  min RTT  avg RTT  max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]       [Mbit/s]   [#/s]     [ms]     [ms]     [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.106 0.157    5.077849    19.37  102.321  102.321  102.321     inf     inf     inf   20 INT_MAX   20    0    0    0    0    0   19    0  59.1    5.9  260.0     open  1448  1500    2     2     0    19    11     0
 0 0.102 0.158    0.022773     0.00      inf      inf      inf   0.017  15.654  58.756   10 INT_MAX    1    0    0    0    0    0    3    0  42.2   19.4  244.0     open  1448  1500    0     0     0     3     0     0
 0 0.157 0.208    2.608870    79.62  144.581  144.590  144.600     inf     inf     inf   29 INT_MAX   32    3    0    0    0    4   19    0  47.6    8.4  248.0 disorder  1448  1500    2     2     0    19    11     2
 0 0.158 0.202    0.021722     0.00      inf      inf      inf   0.019  20.450  37.227   10 INT_MAX    1    0    0    0    0    0    3    0  42.3   14.7  244.0     open  1448  1500    0     0     0     3     0     0
ID  begin    end   through transac  min RTT  avg RTT  max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]  [Mbit/s]   [#/s]     [ms]     [ms]     [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.208 0.251 7.643920  139.96  188.936  188.952  188.970     inf     inf     inf   65 INT_MAX   61    0    0    0    0    0   19    0  45.0    4.8  248.0     open  1448  1500    2     2     0    19    11     2
 0 0.202 0.252 0.031740    0.00      inf      inf      inf   0.749   8.374  21.913   10 INT_MAX    1    0    0    0    0    0    3    0  42.3   11.1  244.0     open  1448  1500    0     0     0     3     0     0
 0 0.252 0.300 0.100675    0.00      inf      inf      inf   0.011   4.570  30.149   10 INT_MAX    1    0    0    0    0    0    3    0  43.4   10.6  244.0     open  1448  1500    0     0     0     3     0     0
ID  begin    end     through transac  min RTT  avg RTT  max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]    [Mbit/s]   [#/s]     [ms]     [ms]     [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.251 0.305 15.595148   18.30  239.062  239.062  239.062     inf     inf     inf  101 INT_MAX  102    1    0    0    0    2   49    0  44.3    2.1  248.0 disorder  1448  1500    3     3     0    49    21     2
 0 0.305 0.353  4.145335  316.26  299.394  299.435  299.477     inf     inf     inf   98      59  179   81    4    4    0   85   49    0  55.9    0.3  256.0  recover  1448  1500    7     7     0    49     2     2
 0 0.300 0.353  0.126349    0.00      inf      inf      inf   0.011   2.058  31.996   10 INT_MAX    1    0    0    0    0    0    3    0  45.5   12.1  248.0     open  1448  1500    0     0     0     3     0     0
 0 0.353 0.407 28.871708  422.19  343.649  343.717  343.782     inf     inf     inf  167 INT_MAX  173    6    0    0    0    8   85    0  52.5   10.6  256.0 disorder  1448  1500    7     7     0    85    75     3
 0 0.353 0.405  0.018503    0.00      inf      inf      inf   0.947   7.341  19.273   10 INT_MAX    1    0    0    0    0    0    3    0  45.3    9.6  248.0     open  1448  1500    0     0     0     3     0     0
 0 0.407 0.452 46.479550  642.73  388.115  388.202  388.290     inf     inf     inf  295 INT_MAX  291    0    0    0    0    0  127    0  44.1    0.9  248.0     open  1448  1500    9     9     0   127    27     7
 0 0.405 0.458  0.231744    0.00      inf      inf      inf   0.011   1.902  41.387   10 INT_MAX    1    0    0    0    0    0    3    0  45.4    7.3  248.0     open  1448  1500    0     0     0     3     0     0
 0 0.452 0.509  6.928995    0.00      inf      inf      inf     inf     inf     inf  294     163  483  189    1    1    0  194  127    0  76.5    0.9  280.0  recover  1448  1500   10    10     0   127    27     9
 0 0.458 0.506  0.291782    0.00      inf      inf      inf   0.011   1.338  54.966   10 INT_MAX    2    0    0    0    0    0    3    0  49.6   13.9  252.0     open  1448  1500    0     0     0     3     0     0
ID  begin    end       through  transac  min RTT  avg RTT  max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]      [Mbit/s]    [#/s]     [ms]     [ms]     [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.509 0.556  107.656510  1749.38  467.111  483.575  498.367     inf     inf     inf  619 INT_MAX  619    0    0    0    0    0  127    0  49.7    0.6  252.0     open  1448  1500   11    11     0   127    70     9
 0 0.506 0.557    0.214658     0.00      inf      inf      inf   0.011   1.491  23.111   10 INT_MAX    3    0    0    0    0    0    3    0  50.2   11.7  252.0     open  1448  1500    0     0     0     3     0     0
 0 0.556 0.604   27.364461   104.39  521.233  521.246  521.260     inf     inf     inf  589     314  787  198    3    3    0  202  127    0  76.7    0.4  280.0  recover  1448  1500   15    15     0   127   164    11
 0 0.557 0.602    0.134967     0.00      inf      inf      inf   0.011   0.784  14.675   10 INT_MAX    2    0    0    0    0    0    3    0  53.6   15.5  256.0     open  1448  1500    0     0     0     3     0     0
 0 0.602 0.651    0.019567     0.00      inf      inf      inf   0.012  29.359  88.044   10 INT_MAX    1    0    0    0    0    0    3    0  60.4   20.2  264.0     open  1448  1500    0     0     0     3     0     0
ID  begin    end       through  transac  min RTT  avg RTT  max RTT min IAT avg IAT max IAT cwnd     ssth uack sack lost retr tret fack reor bkof    rtt rttvar     rto ca state  smss  pmtu cret cfret ctret dupth lreos tdsac
      [s]    [s]      [Mbit/s]    [#/s]     [ms]     [ms]     [ms]    [ms]    [ms]    [ms]  [#]      [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]  [#]   [ms]   [ms]    [ms]            [B]   [B]  [#]   [#]   [#]   [#]   [#]   [#]
 0 0.604 0.652    0.000000   413.27  597.679  597.739  597.800     inf     inf     inf  343     314  904  512   91   42    0  628  127    0  77.8    0.3  280.0  recover  1448  1500   55    55     0   127    88    11
 0 0.652 0.703   70.374231   683.34  641.584  641.656  641.680     inf     inf     inf  314     314  790  476  111  111    0  590  127    0  60.5    4.1  264.0  recover  1448  1500  131   131     0   127    66    13
 0 0.651 0.701    0.353642     0.00      inf      inf      inf   0.011   0.613  16.828   10 INT_MAX    2    0    0    0    0    0    3    0  60.6   15.4  264.0     open  1448  1500    0     0     0     3     0     0
 0 0.701 0.750    0.527293     0.00      inf      inf      inf   0.011   0.451  17.575   10 INT_MAX    3    0    0    0    0    0    3    0  59.7   10.7  260.0     open  1448  1500    0     0     0     3     0     0
 0 0.703 0.752  116.888698  1908.02  659.547  681.967  696.223     inf     inf     inf  283     157  556  273    2    2    0  279  127    0  61.0    0.5  264.0  recover  1448  1500  133   133     0   127     5    14
 0 0.752 0.802   33.837902   258.16  711.121  711.156  711.188     inf     inf     inf  157     157  535  378    4    4    0  384  127    0  58.5    0.3  260.0  recover  1448  1500  137   137     0   127    89    14
 0 0.750 0.805    0.361845     0.00      inf      inf      inf   0.011   1.047  44.815   10 INT_MAX    2    0    0    0    0    0    3    0  59.0    7.3  260.0     open  1448  1500    0     0     0     3     0     0
 0 0.802 0.858   51.779207  1113.31  761.418  761.901  762.599     inf     inf     inf  157     157  427  270    3    3    0  274  127    0  52.5    1.0  256.0  recover  1448  1500  139   139     0   127    21    14
 0 0.805 0.851    0.153389     0.00      inf      inf      inf   0.011   2.543  41.031   10 INT_MAX    1    0    0    0    0    0    3    0  59.7    5.3  260.0     open  1448  1500    0     0     0     3     0     0
```

Figure 4.3: A sample flowgrind logfile output

5 Evaluation and Discussion

This chapter describes the various scenarios under which Linux's reordering detection and reaction mechanism, TCP-NCR and TCP-aNCR are evaluated. In the first section of this chapter, the algorithms are evaluated under lower data rates to understand the behaviour and the pros and cons of the algorithms. In the latter section, the algorithms are evaluated at higher data rates commonly found in data center scenarios.

The algorithms are evaluated for throughput, application perceived delay, number of spurious retransmits under varying parameters of bottleneck bandwidth, round trip time, reordering delay and reordering rate. The experiments are run for 120 seconds (lower data rates) or 11 seconds (higher data rates) and an iteration factor of 10. The timestamps option is disabled for NCR and aNCR to observe the worst case scenario performance. Furthermore, the slow start bug in NCR has been fixed in the experiments to have a fair comparison between NCR and aNCR. There are in-total 6 possible combinations of algorithms which are evaluated in this thesis. They are, Linux with timestamps disabled and SACK enabled (Native Linux DS), Linux with timestamps and SACK enabled (Linux TS), NCR with careful limited transmit (TCP-NCR CF), NCR with aggressive limited transmit (TCP-NCR AG), aNCR with careful limited transmit (TCP-aNCR CF), aNCR with aggressive limited transmit (TCP-aNCR AG).

5.1 Lower data rates

In this section, the algorithms are evaluated under different scenarios present in the wild internet. The low data rate environment is emulated with reference to the *Common TCP Evaluation Suite.*[44] The topology used for the measurements is as described in fig 4.1.

5.1.1 Scenario 1: Performance without packet reordering under different bottleneck bandwidths

In this section we study the performance of Linux, TCP-NCR and TCP-aNCR with no Packet reordering in the network. Without packet reordering in the network, the only reason for triggering ELT is due to network congestion because of our bottleneck node. The experiment is conducted at various values of bottleneck.

With no reordering, TCP-aNCR maintains a duplicate threshold of three packets unlike TCP-NCR which waits for a congestion window worth of packets to decide that the

packet has been lost and not reordered. Thus, TCP-aNCR should be more responsive to network congestion and should immediately retransmit the lost packet. This should be evident by comparing application perceived RTT.

Testbed settings:

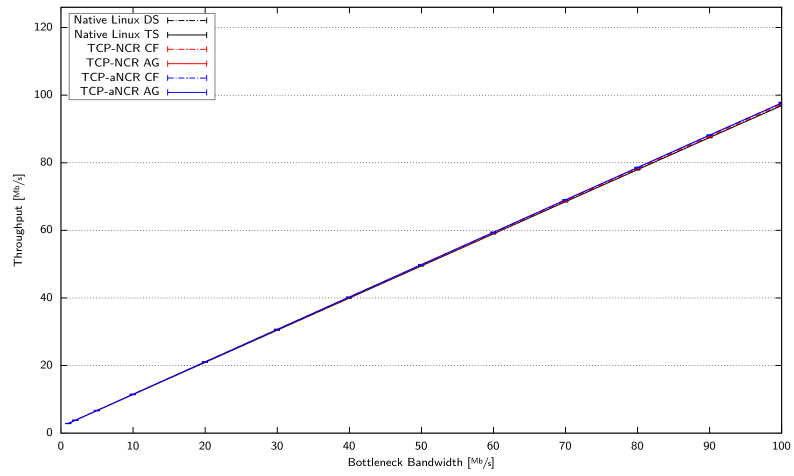
RTT: 40ms
BNBW: 1Mbps - 100Mbps
Reordering rate: 0%

This scenario is studied under:

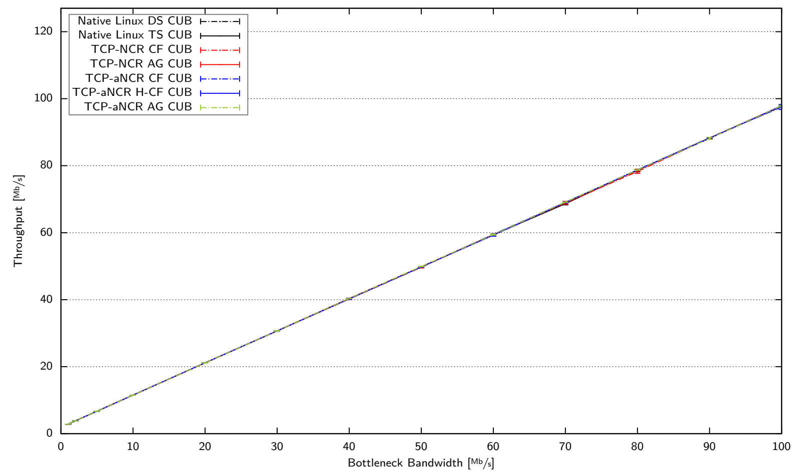
1. Bulk traffic
   a) No cross traffic
      i. With Reno congestion control algorithm [51]
      ii. With CUBIC congestion control algorithm [52]
   b) With Reno and static aNCR dupthresh
   c) with cross traffic
2. Request response traffic and CUBIC

The graphs in fig 5.1a and fig 5.1b are plots of maximum attained throughput over bottleneck bandwidth. The experiments are performed with Reno[51] (fig: 5.1a) and CUBIC[52] (fig 5.1b)because, in most Linux systems, CUBIC is the default congestion control algorithm. we notice that Linux's default algorithm, NCR and aNCR perform equally well in the absence of packet reordering in the network. The dupthresh in this scenario is three for Linux and aNCR. However, for NCR, the dupthresh is approximately equal to the value of congestion window. The experiment is also conducted with dynamic dupthresh turned off for aNCR and with Reno congestion control algorithm[51]. The values of average throughput achieved (fig 5.1c) are similar to those in fig 5.1a and fig 5.1b. There were no spurious retransmissions as there was no packet reordering in the network.

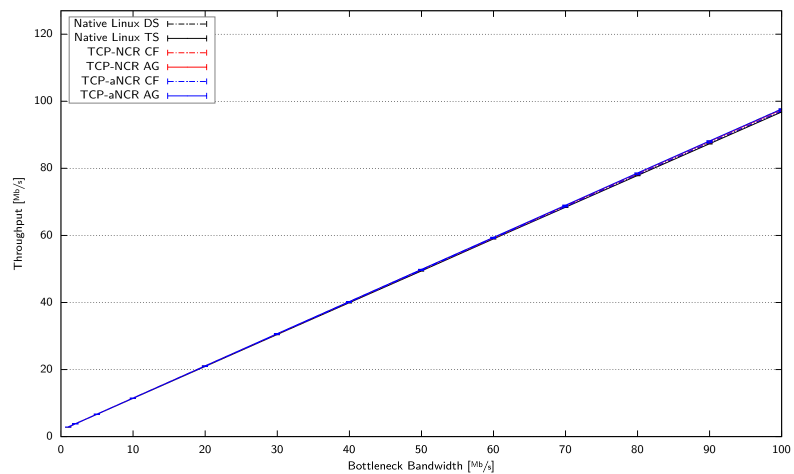
The main purpose of this measurement is to analyse the benefits of latency improvements with aNCR in comparison to NCR because of the variable dupthresh of aNCR. For this purpose, we measure the average application-perceived latency with Reno[51] congestion control algorithm. The average RTT values are tabulated in table 5.1. As we can observe from the table 5.1, at lower data rates, the average application-perceived latency is lesser for Linux and aNCR and slightly higher for NCR. This is because of the lower dupthresh in case of Linux and aNCR in comparison with NCR. This difference in latency is visible at lower data rates and lesser at higher data rates because the experiments were conducted for 120 seconds and with higher bottleneck bandwidths, there are fewer loss-recovery cycles in the 120 second test duration thus, the values are in similar range.



(a) Reno congestion control



(b) CUBIC congestion control



(c) aNCR static dupthresh with Reno congestion control

Figure 5.1: Lower data rate - Scenario 1: Average throughput vs BNBW

BNBW	Linux DS	Linux TS	NCR CF	NCR AG	aNCR CF	aNCR AG	aNCR CF static dupthresh	aNCR AG static dupthresh
1	30.030387	30.027638	30.100013	30.191528	30.027695	30.029036	30.09609	30.04671
2	30.000046	30.017357	30.017225	29.969165	30.006216	30.011953	30.015353	29.961395
5	18.968994	18.971599	19.037482	18.981349	18.968374	18.976077	19.0388	18.966765
10	10.668574	10.671492	10.718232	10.677515	10.666853	10.670589	10.725689	10.673792
20	5.646246	5.644762	5.671485	5.650121	5.648805	5.647219	5.680073	5.646948
30	3.844705	3.841509	3.859232	3.842671	3.840104	3.840494	3.86055	3.840203
40	2.911151	2.909424	2.923892	2.911188	2.90943	2.90847	2.925749	2.910272
50	2.341407	2.339534	2.350968	2.341196	2.339963	2.339129	2.352693	2.338962
60	1.958057	1.956865	1.966179	1.958889	1.957811	1.957055	1.967908	1.957342
70	1.681853	1.682081	1.689619	1.683049	1.68103	1.681325	1.692674	1.681968
80	1.474749	1.474945	1.481854	1.475548	1.47464	1.474166	1.482659	1.47383
90	1.313329	1.312778	1.319057	1.313571	1.311865	1.313	1.320012	1.312407
100	1.184563	1.183705	1.189476	1.18395	1.183794	1.183983	1.190812	1.183753

Table 5.1: Scenario 1: Average application-perceived latency (seconds) vs BNBW (mbps) - Reno

Furthermore, there is a similarity in the latency values of NCR and aNCR with static dupthresh because, when dynamic dupthresh is disabled in aNCR, the aNCR's dupthresh is equal to NCR dupthresh.

The algorithms were also evaluated with cross-traffic passing through the bottleneck node. The cross traffic generator and sink were used as shown in fig 4.1. The behaviour of reordering algorithms does not depend based on the presence of cross traffic or concurrent flows. This is because, a packet reordering component in the network affects all the flows and the occurrence of packet reordering in one flow does not affect the other flow in any way. The presence of cross traffic just decreased the average throughput of our experimental flow. The cross traffic generator generated a standard TCP flow with Reno congestion control algorithm with SACK option enabled.

The algorithms are also evaluated under request-response traffic. The average number of transactions per second at various bottleneck bandwidths are tabulated as in table 5.2. We notice that all the reordering algorithms perform equally well under no packet reordering in the network.

5.1.2 Scenario 2: Performance with packet reordering under different bottlenecks

In this scenario, we compare the performance of Linux, TCP-NCR and TCP-aNCR under different bottlenecks varying from 1Mbps upto 100Mbps. We introduce a reordering percentage of 2% in the network with and a reordering delay of 20ms. A good reordering algorithm should be robust in the presence of packet reordering in the network.

Testbed settings:

BNBW	Linux DS	Linux TS	NCR CF	NCR AG	aNCR CF	aNCR H-CF	aNCR AG
1	341	341	342	340	342	342	341
2	688	687	688	688	688	688	688
5	1718	1718	1718	1716	1717	1717	1718
10	3433	3436	3434	3436	3437	3436	3435
20	6860	6862	6864	6865	6865	6869	6865
30	10231	10266	10290	10267	10275	10241	10259
40	13717	13730	13726	13726	13660	13726	13623
50	17062	17094	17126	17099	17145	17123	17128
60	20561	20533	20572	20562	20556	20562	20555
70	24001	24003	23953	24015	23977	23898	23899
80	27427	27442	27433	27428	27299	27222	27343
90	30816	30857	30767	30698	30871	30839	30871
100	34250	34278	34081	34289	34224	34193	34115

Table 5.2: Scenario 1: Average transactions/s vs bottleneck bandwidth (mbps) - CUBIC

RTT: 40ms
BNBW: 1Mbps - 100Mbps
Reordering rate: 2%
Reordering delay: 20ms

This scenario is studied under:

1. Bulk traffic with no cross traffic
   a) With Reno congestion control algorithm [51]
   b) With CUBIC congestion control algorithm [52]
   c) With Reno and static aNCR dupthresh
2. Request response traffic and CUBIC

The plots in fig 5.2a and fig 5.2b are those of average throughput versus bottleneck bandwidth with Reno and CUBIC respectively. The experiment is also conducted with dynamic dupthresh turned off for aNCR and with Reno congestion control algorithm[51]. The values of average throughput achieved (fig 5.1c) are similar to those in fig 5.1a and fig 5.1b.

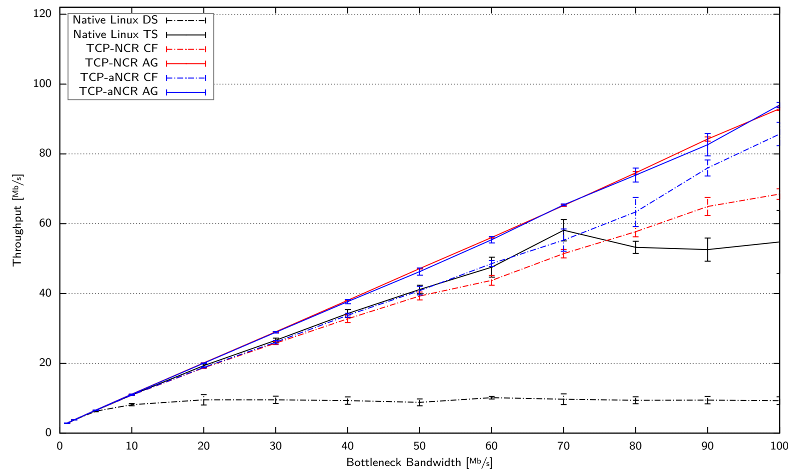
We can see that Linux's performance is poorest among the reordering algorithms. This can be explained by the implementation of Linux's reordering detection and reordering reaction algorithm. Each time a hole is filled in the scoreboard, Linux calculates the reordering extent and sets the maximum reordering extent seen as the dupthresh value. However, the amount of reordering extent Linux can detect and calculate is limited to 127 packets (at the time of implementation). However, in our experiments, with our reordering emulation with NetEm, we found that, even with a constant 2% reordering

rate, reordering extent often exceeded the value of 127. This is more evident at higher data rates. Furthermore, Linux tries to undo recovery (If in Recovery phase) each time there is a DSACK block that acknowledges already acknowledged data (below the cumulative ACK line). However, in most cases, the TCP connection is already in OPEN state or another DISORDER state. Furthermore, Linux kernel does not take into account the reordering extent when a DSACK arrives. With the timestamps option enabled in the Linux kernel, we see an improvement in the performance with Linux's reordering detection and reaction and reno congestion control algorithm. However, as earlier mentioned, at higher data rates, the connection is in persistent reordering phase and does not increase its CWND. Thus, the data rate remains approximately constant after 70 mb/s. With timestamp option enabled and CUBIC congestion control algorithm (fig 5.2b), the performance is as bad as that of timestamps disabled. The reason for the poor performance should be analysed.

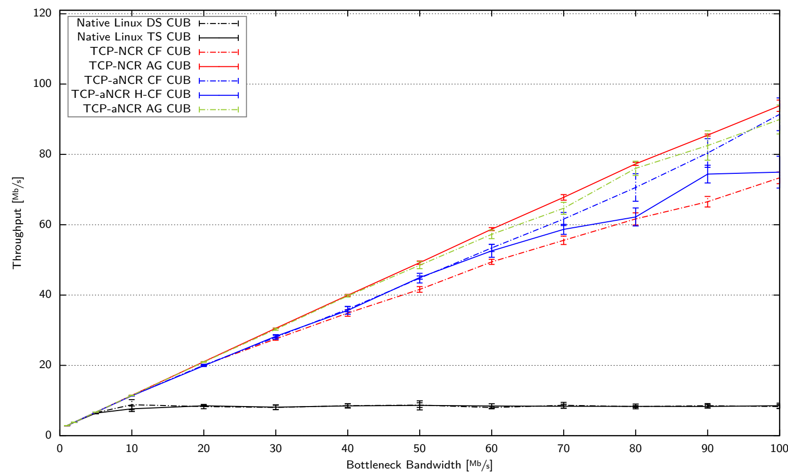
NCR with careful ELT also suffers under packet reordering, especially at higher data rates because, NCR fails to update the congestion window in ELT 3.2.1 and at higher data rates, the number of reordered packets are more and the connection is in persistent reordering state. In this state, NCR fails (STEP T.4) to update its congestion window and therefore performs poorly compared to aNCR. This explains the deviation at 60mb/s and beyond. NCR and aNCR had the least number of spurious retransmissions at approximately 10 data segments. The application perceived latency was the least with NCR and aNCR and the highest with Linux without timestamps.
The algorithms are also evaluated under request-response traffic with CUBIC. The average number of transactions per second at various bottleneck bandwidths are tabulated as in table 5.3. As expected, with NCR and aNCR the transaction rate is higher compared to Linux as explained in the earlier plots. The performance of the algorithms with request-response traffic are in accordance with those of bulk traffic scenarios.

BNBW	Linux DS	Linux TS	NCR CF	NCR AG	aNCR CF	aNCR H-CF	aNCR AG
1	334	341	343	341	341	343	340
2	684	684	686	687	687	687	687
5	1622	1605	1708	1714	1707	1710	1716
10	2143	2761	3373	3418	3348	3384	3415
20	2364	3996	6455	6837	6535	6392	6767
30	2331	2264	9257	10218	9332	9313	10166
40	2265	2153	11800	13602	12307	12014	13084
50	2238	2272	14244	17035	15255	15387	16809
60	2073	2462	16619	20381	17633	18139	19233
70	2444	2698	19334	23694	20709	19636	23430
80	2084	2333	21954	26670	24541	22199	25430
90	2501	2127	23489	29693	25490	24287	27793
100	2168	2336	25327	32979	31311	27149	30907

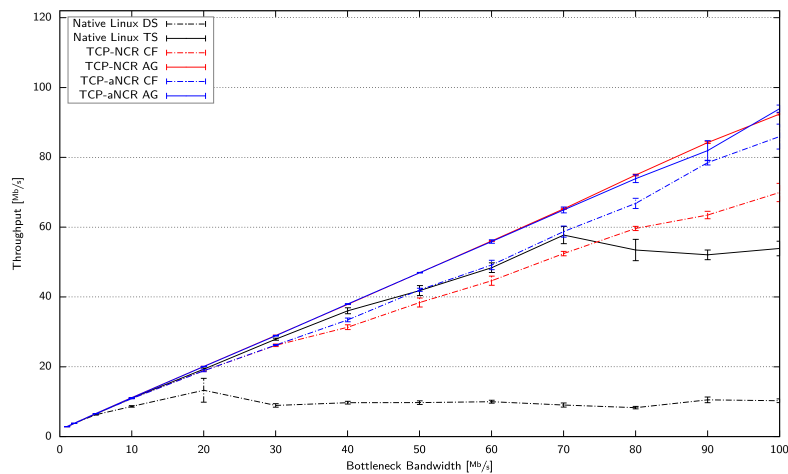
Table 5.3: Scenario 2: Average transactions/s vs bottleneck bandwidth (mbps) - CUBIC



(a) Reno congestion control



(b) CUBIC congestion control



(c) aNCR static dupthresh with Reno

Figure 5.2: Lower data rate - Scenario 2: Average throughput vs BNBW

5.1.3 Scenario 3: Performance with packet reordering under varying RTTs

In this scenario, we compare the performance of Linux, TCP-NCR and TCP-aNCR under varying values of RTT from 5ms to 150ms. We introduce a reordering percentage of 2% in the network with and a reordering delay of 5ms.

The purpose of this experiment is to study the effect of RTT on the performance reordering algorithms. This scenario can be considered as a specific case of scenario 2 with varying RTT. An arbitrary bottleneck of 20 mbps was chosen for this scenario since all the reordering algorithms (except Linux with timestamps disabled) performed better with Reno in the previous scenario.

Testbed settings:

BNBW: 20mb/s
Reordering rate: 2%
Reordering delay: 5ms

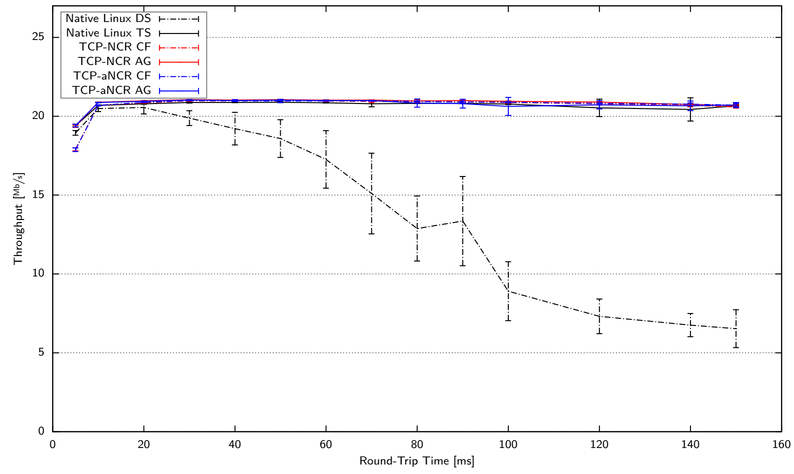
This scenario is studied under:

1. Bulk traffic with no cross traffic
    a) With Reno congestion control algorithm [51]
    b) With Reno and static aNCR dupthresh
2. Request response traffic and CUBIC

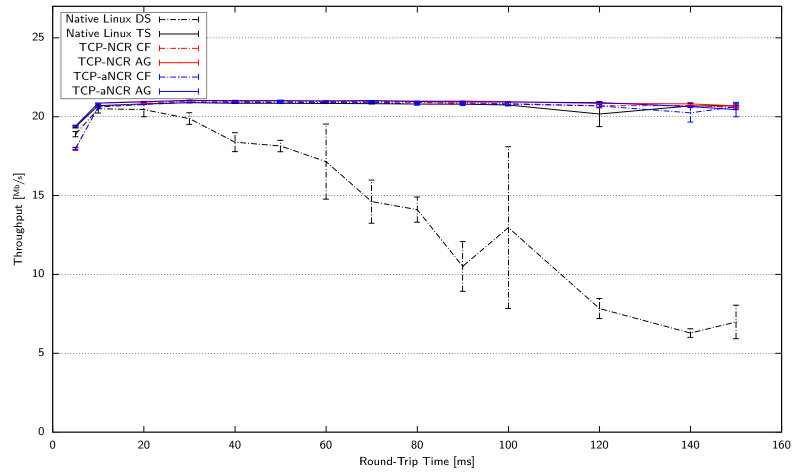
The plots of average transmission rate over RTT in fig 5.3a and fig 5.3b illustrate that the reordering algorithms except for Linux with timestamsp disabled, exhibit robustness against mild packet reordering of 2% and a reordering delay of 5ms. The Linux with timestamp option disabled continues to deteriorate at higher values of RTT. This is because the connection is more often in recovery phase and as explained in scenario 2. The plot in fig 5.12 of spurious retransmits versus RTT values further illustrates the constant transit to recovery phase when timestamp option is disabled in the Linux kernel. Furthermore, with higher RTTs, the congestion window does not increase fast and therefore the transmission rate further deteriorates. The transmission rate directly impacts the application perceived latency of the network. This is illustrated by the plots of application perceived latency versus RTT in fig 5.4a and fig 5.4b. The experiments with request response traffic exhibit the expected behaviour of having similar values as in table 5.3. The spurious retransmits were found to be none with NCR , approximately 4 with aNCR and Linux with timestamps and more than 200 for Linux with timestamps disabled.

5.1.4 Scenario 4: Performance with varying reordering rate

In this section we study the performance of Linux, TCP-NCR and TCP-aNCR under varying reordering rate from 0% upto 40% and a constant reordering delay of 20ms. The

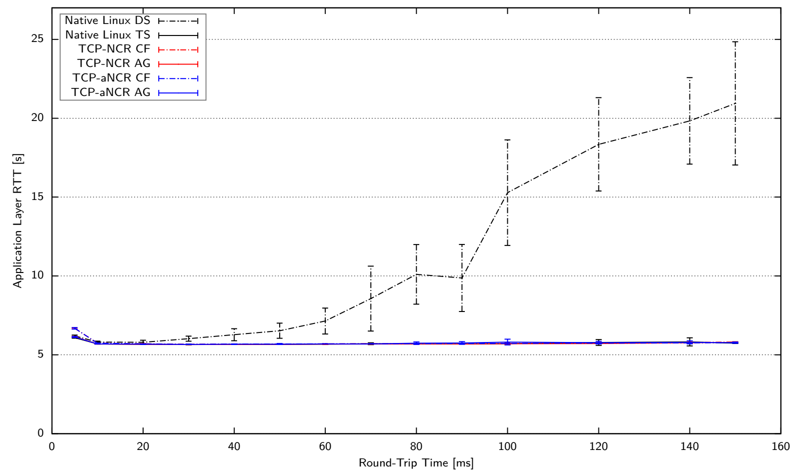


(a) Reno congestion control

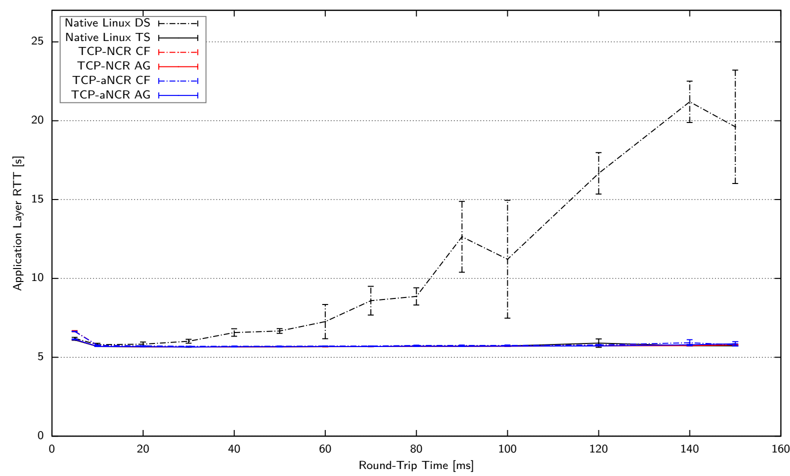


(b) aNCR static dupthresh with Reno

Figure 5.3: Lower Data rate - Scenario 3: Average throughput vs delay



(a) Reno congestion control



(b) aNCR static dupthresh with Reno

Figure 5.4: Lower data rate - Scenario 3: Average application-perceived latency vs delay

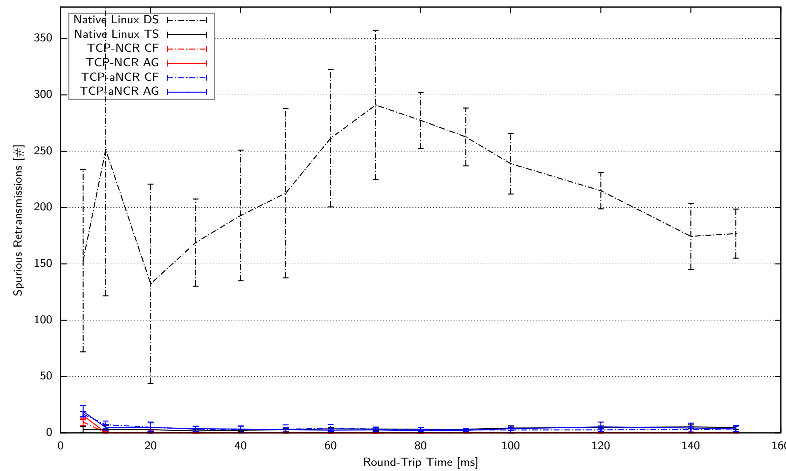


Figure 5.5: Lower data rate - Scenario 3: Spurious retransmits at various RTT values

goal of this scenario is to study the impact of reordering rate on throughput, latency and spurious retransmissions at different reordering rates.

Testbed settings:

RTT: 40ms
Reordering rate: 0% - 40%
Reordering delay: 20ms

This scenario is studied under:

1. Bulk traffic with no cross traffic
   a) With Reno congestion control algorithm [51]
   b) With Reno and static aNCR dupthresh
2. Request response traffic and CUBIC

The plots in fig 5.6a, fig 5.6b and fig 5.6c illustrate the average throughput achieved by the reordering algorithms at various reordering rates. TCP-aNCR performs well at all reordering rates. TCP-NCR with careful limited transmit begins to lose performance when the packet reordering rate is about 3% and from then on, it further begins to lose performance at higher reordering rates. TCP-NCR with aggressive limited transmit begins to lose performance at about 30% reordering rate. Linux on the other hand loses performance sharply when the reordering rate is about 3% and from then on, the throughput begins to improve and reaches the full bottleneck bandwidth when the reordering rate is about 40%. On the other hand, Linux with timestamps enabled behaves similar to that of without timestamps. However, the descend in the throughput is less drastic as that of Linux with timestamps disabled.

The application perceived latency increases with Linux's native algorithm with timestamps disabled. It is at its highest between 2% and 5% and gradually improves as the reordering rate increases. This is the same for NCR and aNCR but is not as drastic as that of Linux without timestamps.

The plot in fig 5.7 highlights the extent of spurious retransmissions across various algorithms. During the reordering rates of 2% to 5% the Linux kernel exhibits high amount of spurious retransmissions because of falsely detecting a reordered packet as lost. Therefore the CWND never grows to utilise the full available bandwidth and are the reasons for its poor throughput and high amount of spurious retransmits. For the other algorithms, the spurious retransmissions were low at about 10 - 20 packets.

The performance also reflects in the results of the request response test (table 5.4) with Linux performing badly at reordering rates between 2% to 5%. NCR and aNCR performs well and have higher transactions/s with NCR's performance going down at higher reordering rates.

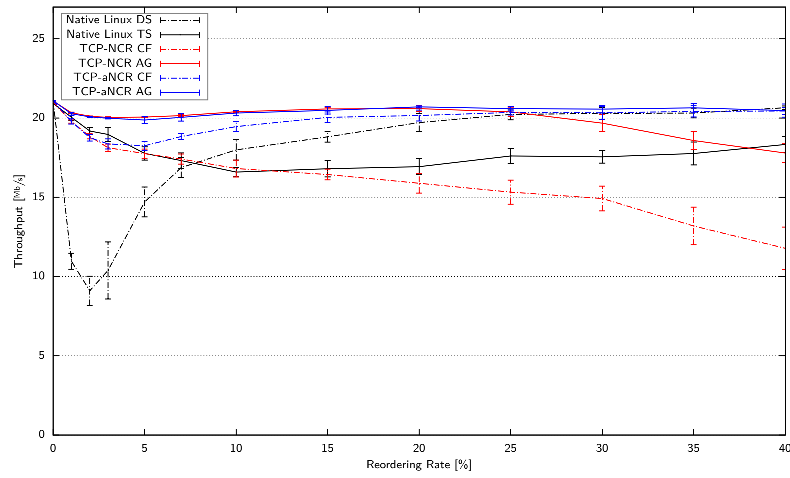
Reordering rate (%)	Linux DS	Linux TS	NCR CF	NCR AG	aNCR CF	aNCR H-CF	aNCR AG
0	6865	6860	6867	6870	6866	6865	6868
1	3008	3065	6682	6837	6683	6600	6833
2	2296	2492	6411	6825	6430	6466	6786
3	2050	2363	6112	6811	6395	6282	6765
5	4424	4308	5905	6807	6409	6273	6728
7	5176	4539	5730	6799	6465	6479	6781
10	5276	5404	5326	6799	6644	6629	6719
15	5998	6187	4946	6711	6723	6623	6752
20	6376	6418	4658	6560	6797	6733	6733
30	6230	6473	4393	6258	6709	6749	6840
35	6375	6339	4135	5889	6656	6819	6691
25	6699	6453	3059	5479	6811	6717	6679
40	6538	6420	2935	5227	6758	6713	6837

Table 5.4: Scenario 4: Average transactions/s vs reordering rate - CUBIC

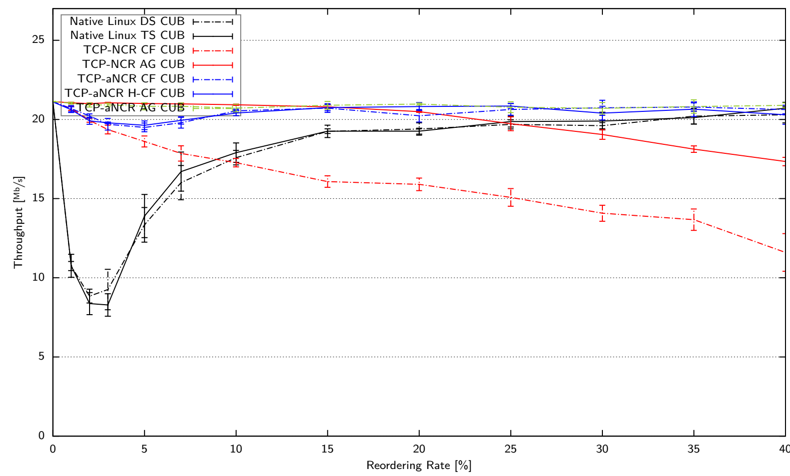
5.1.5 Scenario 5: Performance with varying reordering delay

In this section we study the performance of TCP-NCR and TCP-aNCR under varying reordering delay from 5ms upto 100ms and a constant RTT of 40ms. The goal of this scenario is to study the throughput, latency and spurious retransmissions at different reordering rates.

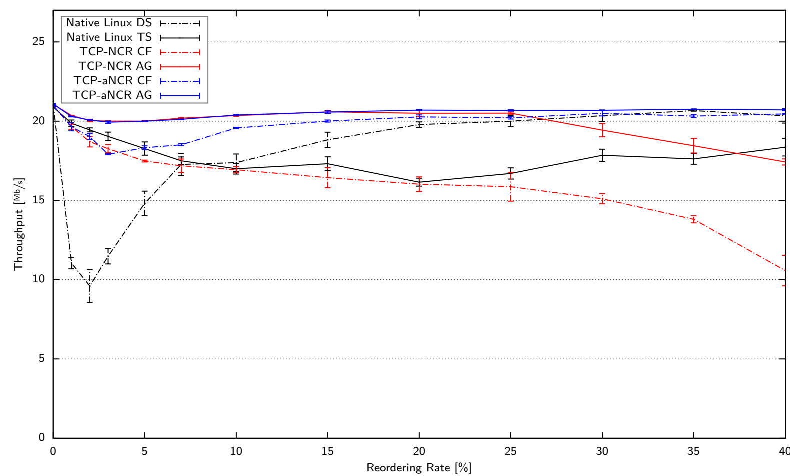
With a varying reordering delay, we are also varying the relative reordering extent. At higher reordering delays, aNCR's reordering extent reaches its maximum factor of 1,



(a) Reno congestion control



(b) CUBIC congestion control



(c) aNCR static dupthresh with Reno

Figure 5.6: Lower data rate - Scenario 4: Average throughput vs Reordering rate

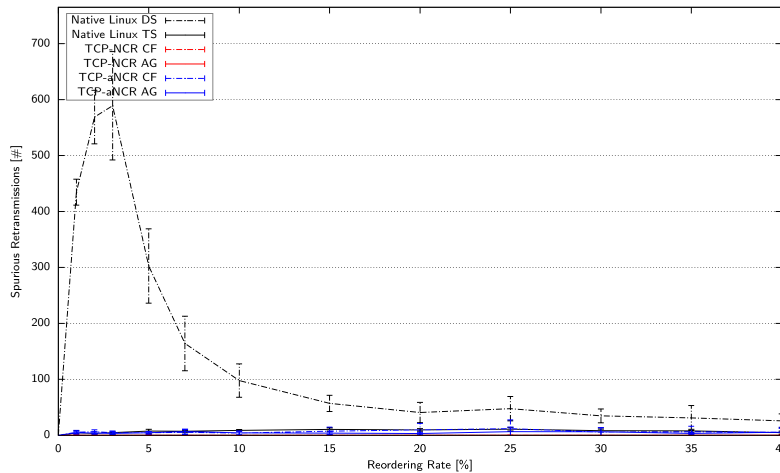


Figure 5.7: Lower data rate - Scenario 4: Spurious retransmits at various RTT values

i.e., equals the duplicate threshold value of NCR.

Testbed settings:

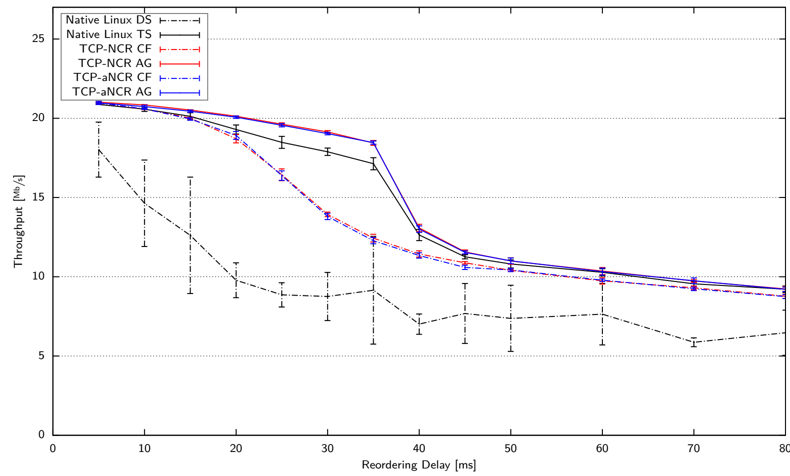
RTT: 40ms
Reordering rate: 2%
Reordering delay: 5ms - 100ms

This scenario is studied under:

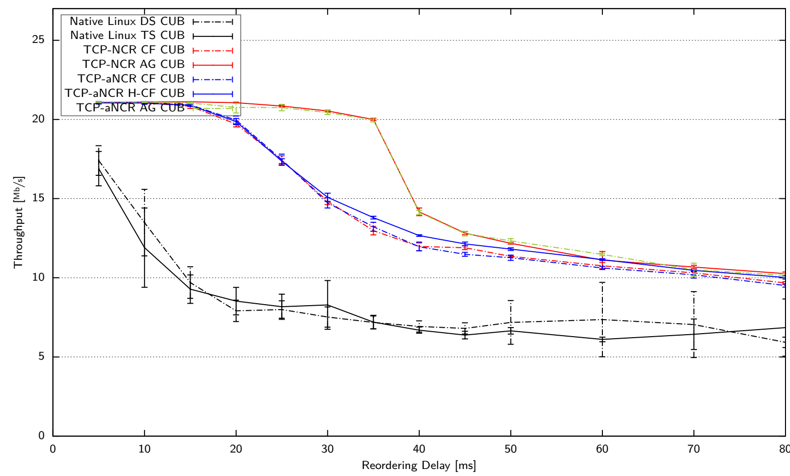
1. Bulk traffic with no cross traffic
   a) With Reno congestion control algorithm [51]
   b) With Reno and static aNCR dupthresh
2. Request response traffic and CUBIC

The plots in fig 5.8a, fig 5.8b and fig fig:s5bdrbnbw of average throughput over reordering delay demonstrates the performance of various algorithms. The algorithms are measured with Reno, CUBIC and with aNCR static dupthresh. Linux without timestamps performs the poorest. From 0ms to 35 ms of reordering delay, The (a)NCR versions with aggressive limited transmit have a higher average throughput. Beyond 35 ms, the reordered packets are no more under the 1 CWND limit that NCR and aNCR have and therefore, cannot detect reordered packets beyond this value. Thus, the throughput drops drastically at about 35ms. The behaviour is similar with Reno and CUBIC congestion control. The application perceived latency increases beyond 35ms for NCR and aNCR.

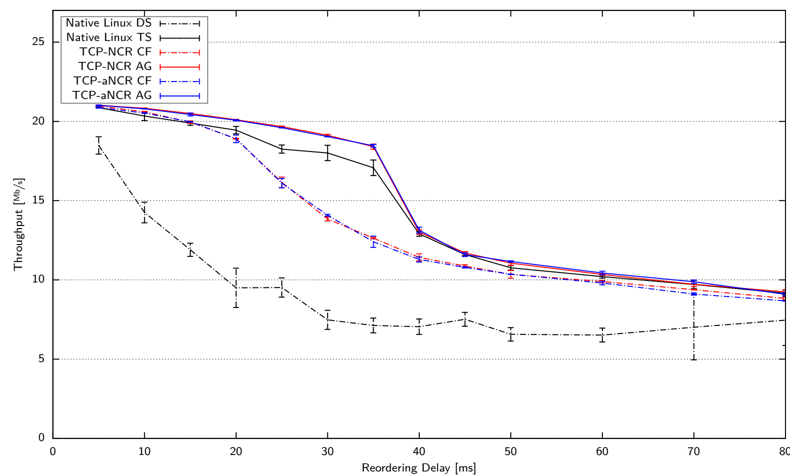
The plot of average spurious transmits versus reordering delay in fig 5.9 clearly illustrates this inability of the algorithms by the sharp increase in the spurious retransmits at about 35ms.



(a) Reno congestion control



(b) CUBIC congestion control



(c) aNCR static dupthresh with Reno

Figure 5.8: Lower data rate - Scenario 5: Average throughput vs Reordering delay

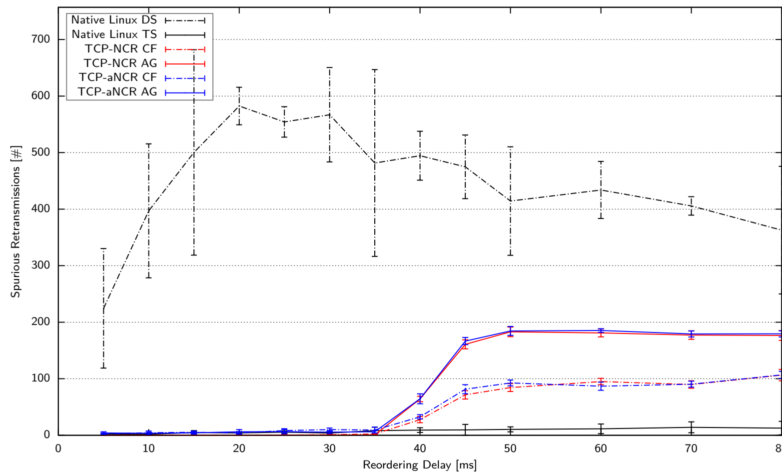


Figure 5.9: Lower data rate - Scenario 5: Spurious retransmits at various reordering delays

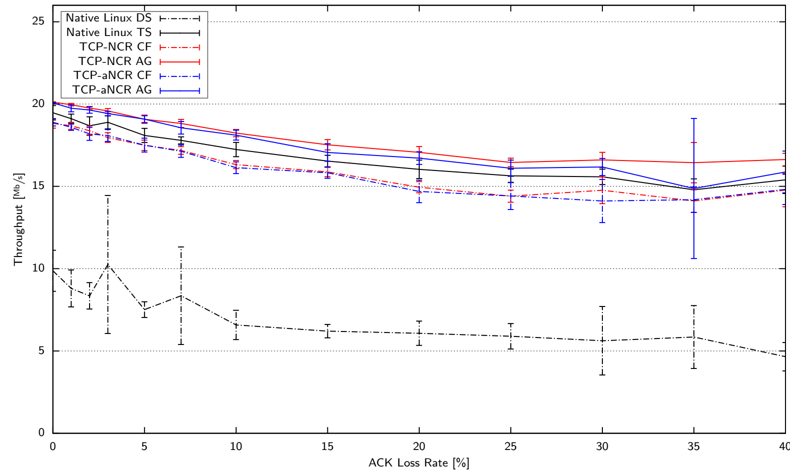
The request response tests as tabulated in 5.5 also agree with the bulk transfer tests with average number of transaction slightly decreasing between 0ms and 40ms of reordering delay and drastically decreasing beyond 40ms of reordering delay.

Reordering delay	Linux DS	Linux TS	NCR CF	NCR AG	aNCR CF	aNCR H-CF	aNCR AG
5	4497	4773	6853	6854	6860	6827	6849
10	3094	3624	6844	6859	6804	6827	6822
15	2788	2421	6777	6850	6778	6713	6851
20	2026	2109	6413	6808	6522	6400	6784
25	2158	2011	5246	6748	5318	5291	6718
30	2045	2022	4570	6630	4470	4608	6547
35	2007	1856	3930	6341	3943	4196	6343
40	1826	1670	3644	4275	3681	3818	4278
45	1495	1552	3426	3836	3416	3614	3816
50	1577	1566	3189	3661	3352	3489	3639
60	1673	1447	3112	3314	3117	3267	3384
70	2031	2035	2891	3092	2860	3001	3108
80	1994	1966	2618	2853	2666	2825	2846

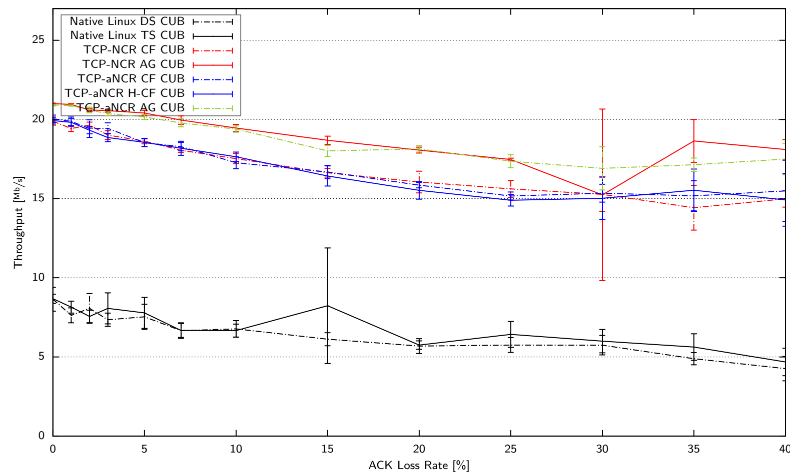
Table 5.5: Scenario 5: Average transactions/s vs reordering delay - CUBIC

5.1.6 Scenario 6: Performance with reverse path loss

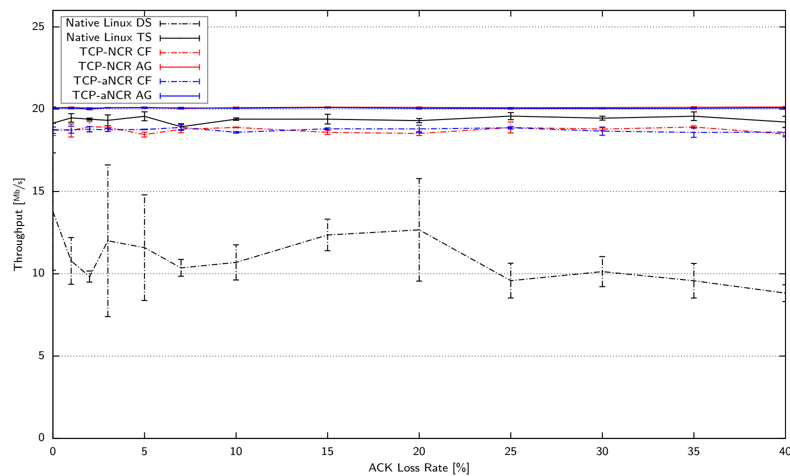
In this section we study the performance of Linux, TCP-NCR and TCP-aNCR under reverse path losses and packet reordering. aNCR depends on DSACKs to detect spurious retransmissions. In the event of finding a DSACK, aNCR restores the previous relative reordering extent. However, DSACKs are reported in only 1 acknowledgement. Therefore, aNCR may be vulnerable to high reverse path losses.



(a) Reno congestion control



(b) CUBIC congestion control



(c) aNCR static dupthresh with Reno

Figure 5.10: Lower data rate - Scenario 6: Average throughput vs Reordering delay

On the contrary, aNCR implements burst protection during extended limited transmit. In the event of a duplicate ACK with a SACK block acknowledging more than one segment, aNCR ensures that only IW worth of packets are transmitted. Therefore, with high reverse path losses and when in extended limited transmit, we can observe the reduction in the throughput with aNCR in reaction to the congested network.

Testbed settings:

RTT: 40ms
Reordering rate: 2%
Reordering delay: 20ms
Reverse path loss: 0% - 30%

This scenario is studied under:

1. Bulk traffic with no cross traffic
    a) With Reno congestion control algorithm [51]
    b) With Reno and static aNCR dupthresh
2. Request response traffic and CUBIC

The plots in fig 5.10a, fig 5.10b and fig 5.10c show the average throughput of the algorithms at various different ACK loss rates. NCR is not dependent on DSACKs and therefore exhibits a better performance. aNCR is dependent on DSACKs and therefore with ACK losses and the loss of crucial DSACKs, aNCR is unable to restore the values of relative and absolute reordering extents when operating with timestamps option disabled and with dynamic dupthresh. As we see in fig 5.10c, aNCR performs on par with NCR with static dupthresh and is more robust to ACK loss.

The request-response tests indicate that NCR and aNCR have the best transaction rate and reflects the results of bulk traffic with NCR slightly performing better than aNCR. The spurious retransmissions are less than 10 for Linux with timestamps, NCR and aNCR. With timestamps disabled, the spurious retransmissions is about 500 data segment for Linux.

5.2 Data center environment

In this section, we study the performance of Linux, TCP-NCR and TCP-aNCR at high data rates. The data rate environment is setup with reference to *Common TCP Evaluation Suite.*[44], *Understanding Data Center Traffic Characteristics*[53] and *Network traffic characteristics of data centers in the wild.*[54] In our test bed, the following experiments are conducted over 1GbeE and 10GbE links as described in fig 4.2. There

are no traffic shaping or traffic policing in the following experiments. The scenarios are studied under the presence and absence of cross traffic. During our tests, it was found that the existence of cross traffic was not affecting the performance of the reordering algorithms. The throughput of the flows were found lesser with cross traffic.

5.2.1 Scenario 1: Performance without packet reordering.

In this scenario, we study the performance of the algorithms under no packet reordering. This is similar to scenario 1 at lower data rates. The experiment was conducted with Reno congestion control [51].
Testbed settings:

RTT: 0.3ms (measured)
bottleneck link capacity: 10Gbps
Reordering rate: 0%
Reordering delay: 0ms

All the algorithms performed equally well with reaching atleast 9.3 Gbps. Linux with timestamps disabled attained an average throughput of 9.36 Gbps and with timestamps enabled 9.33 Gbps as illustrated in fig 5.11. TCP-NCR with careful limited transmit had an average throughput of 9.39 Gbps and aggressive limited transmit 9.57 Gbps. TCP-aNCR with careful and aggressive limited transmit had an average throughput of 9.33 Gbps. There were no significant differences found in the latency values among algorithms.

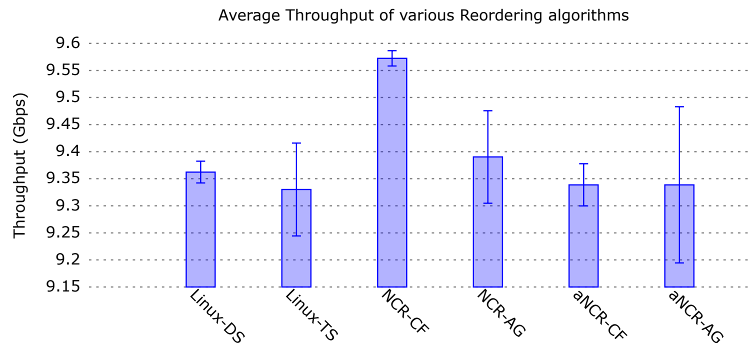


Figure 5.11: High data rate: Performance of algorithms under no packet reordering

5.2.2 Scenario 2: Performance under mild packet reordering.

In this section, we study Linux, TCP-NCR and TCP-aNCR with mild Packet reordering in the network. We introduce a mild packet reordering of 2% and a reordering delay of

0.2ms. The scenario is similar to scenario 2 at lower data rates.

Testbed settings:

RTT: 0.3ms (measured)
bottleneck link capacity: 10Gbps
Reordering rate: 2%
Reordering delay: 0.2ms

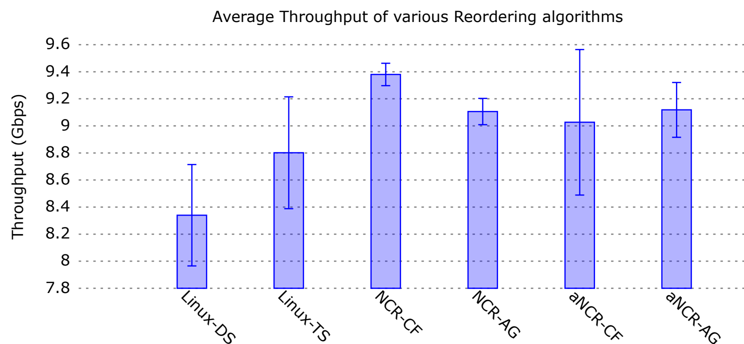


Figure 5.12: High data rate: Performance of algorithms under mild packet reordering

It can be seen that both NCR and aNCR perform well under packet reordering. It is to be noted that the slow start bug in NCR has been fixed to have a fair comparison amongst algorithms. With the fixes, NCR performs the best.

5.2.3 Scenario 3: Performance under varying reordering rates.

In this section, we study Linux, TCP-NCR and TCP-aNCR with varying Packet reordering rate in the network. We vary the reordering rate from 0% to 10% with a reordering delay of 0.2ms. The scenario is similar to scenario 4 at lower data rates. As we can see from fig 5.13 NCR and aNCR perform better under various reordering rates.

Testbed settings:

RTT: 0.3ms
bottleneck link capacity: 1Gbps
Reordering rate: 0% - 10%
Reordering delay: 0.2ms

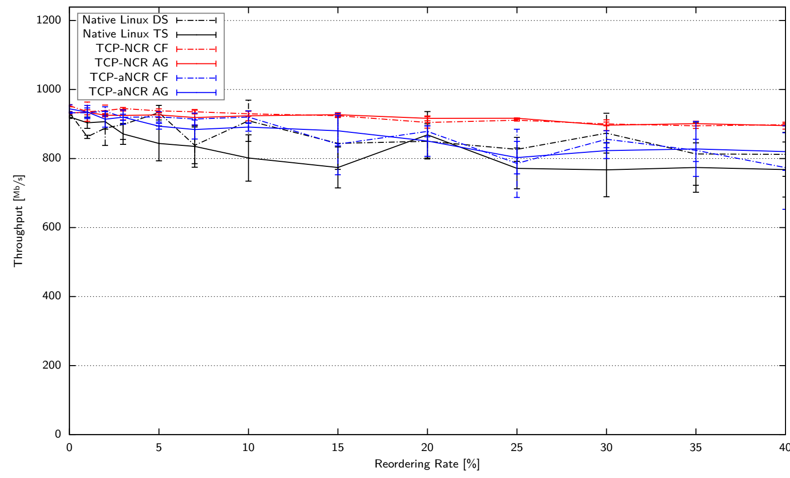


Figure 5.13: High data rate - scenario 3: Average throughput at various reordering rates

6 Conclusion and future work

6.1 Summary

This thesis, demonstrates the inherent problem with TCP in handling packet reordering, the various disadvantages of packet reordering are highlighted. Various approaches to make TCP robust against packet reordering are studied. Furthermore, this work evaluates Linux v3.16 , TCP-NCR and TCP-aNCR's performance against packet reordering under higher and lower data rates; under bulk traffic and request-response traffic.

The experiments prove that TCP-aNCR algorithm is robust against packet reordering as it performs well under all scenarios of packet reordering without latency trade-offs. As earlier described in 5, the slow start bug in NCR was fixed during the evaluation to have a fair comparison among all the algorithms. TCP-aNCR's adaptable DUPTHRESH is a very important aspect of the algorithm as it is able to maintain a value of 3 in case of no packet reordering in the connection and is able to dynamically adapt the DUPTHRESH in the face of reordering. Additionally, with the calculation of relative reordering extent, TCP-aNCR is able to accurately calculate the value of DUPTHRESH in case of varying data rate in the connection as absolute reordering extent is prone to sender's data rate.

Furthermore, TCP-aNCR can be easily integrated in the Linux kernel as most of the functionality is already present in the kernel. There have been many recent modifications being done to the Linux kernel to make it robust against packet reordering [55]. However, Linux's approach is still based on absolute reordering extent which makes it still vulnerable to packet reordering because of its aforementioned disadvantages. Moreover, the maximum absolute reordering extent value is capped at 300 which limits the increase of DUPTHRESH in the presence of reordering to 300. Though it has been exposed as a sysctl [56] parameter making it tunable, it is a system wide setting and may not make sense in case of multipath TCP or the user may not at all make use of this feature. However, with TCP-aNCR no such user tunable parameter is needed as it is able to adapt its DUPTHRESH pro-actively.

6.2 Future work

TCP-aNCR has performed well under various scenarios which emulated in the testbed set up. However, It needs to be emulated in wide range of other network environments such as satellite links, cellular networks and real data center networks. During the thesis work, for higher data rate evaluation, It needed to be switched back to 1GbE from

10GbE as it was found that NetEm was not able to process reordering of packets at very high data rates. Thus, for high data rate scenarios, it should be evaluated under a set up which already has some other forms of packet reordering.

Furthermore, the option of having a dynamic DUPTHRESH in aNCR is a good feature, further research needs to be done as to under which network scenarios, it is most beneficial. Extended limit transmit needs allocation of more buffer to accommodate new data sent during DISORDER which is another area that needs further research. Furthermore, the optimal limited transmit among careful limited transmit, aggressive limited transmit and hybrid limited transmit needs to be done as it was left for discussion by the original authors [2]. One of the interesting observations which came across during the evaluation was Linux with timestamps enabled performed poorly under packet reordering with CUBIC congestion control. Hence it would be interesting to evaluate the performance impact of packet reordering with various other congestion control algorithms as well to strengthen the need for a reordering robust TCP.

Bibliography

[1] J. Postel, "Transmission Control Protocol." RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[2] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events." RFC 4653 (Experimental), Aug. 2006.

[3] A. Hannemann, A. Zimmermann, C. Wolff, and L. Schulte, "Detection and Quantification of Packet Reordering with TCP,"

[4] A. Hannemann, A. Zimmermann, C. Wolff, and L. Schulte, "Making TCP Adaptively Robust to Non-Congestion Events,"

[5] T. linux foundation, "Netem." `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`, 2009. [Online; accessed Dec-2014].

[6] H. P. P. Fabio Ludovici, "Net-em." `http://man7.org/linux/man-pages/man8/tc-netem.8.html`, 2011. [Online; accessed Dec-2014].

[7] A. Zimmermann, A. Hannemann, and T. Kosse, "Flowgrind - a new performance measurement tool," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pp. 1–6, Dec 2010.

[8] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms." RFC 2001 (Proposed Standard), Jan. 1997. Obsoleted by RFC 2581.

[9] E. Blanton and M. Allman, "On making tcp more robust to packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 32, pp. 20–30, Jan. 2002.

[10] M. Zhang and B. Karp, "Rr-tcp: A reordering-robust tcp with dsack," pp. 95–106, 2003.

[11] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser, "Packet Reordering Metrics." RFC 4737 (Proposed Standard), Nov. 2006. Updated by RFC 6248.

[12] J. Bennett, C. Partridge, and N. Shectman, "Packet reordering is not pathological network behavior," *Networking, IEEE/ACM Transactions on*, vol. 7, pp. 789–798, Dec 1999.

[13] V. Paxson, "End-to-end internet packet dynamics," *Networking, IEEE/ACM Transactions on*, vol. 7, pp. 277–292, Jun 1999.

[14] Y. Wang, G. Lu, and X. Li, "A study of internet packet reordering," in *Information Networking. Networking Technologies for Broadband and Mobile Networks* (H.-K. Kahng and S. Goto, eds.), vol. 3090 of *Lecture Notes in Computer Science*, pp. 350–359, Springer Berlin Heidelberg, 2004.

[15] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Measurement and classification of out-of-sequence packets in a tier-1 ip backbone," *Networking, IEEE/ACM Transactions on*, vol. 15, pp. 54–66, Feb 2007.

[16] L. Gharai, C. Perkins, and T. Lehman, "Packet reordering, high speed networks and transport protocol performance," in *Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on*, pp. 73–78, Oct 2004.

[17] *Transmission Control Protocol (TCP) Parameters.* Available at `http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1`, updated on: 2014-10-12.

[18] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance." RFC 1323 (Proposed Standard), May 1992. Obsoleted by RFC 7323.

[19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options." RFC 2018 (Proposed Standard), Oct. 1996.

[20] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP." RFC 2883 (Proposed Standard), July 2000.

[21] M. Kühlewind, S. Neuner, and B. Trammell, "On the state of ecn and tcp options on the internet," in *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, (Berlin, Heidelberg), pp. 135–144, Springer-Verlag, 2013.

[22] *Alexa: An Amazon.com company.* Available at `http://www.alexa.com`, `http://aws.amazon.com/awis/`.

[23] M. Laor and L. Gendel, "The effect of packet reordering in a backbone link on application throughput," *Network, IEEE*, vol. 16, pp. 28–36, Sep 2002.

[24] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, (New York, NY, USA), pp. 314–329, ACM, 1988.

[25] A. Dixit, P. Prakash, Y. Hu, and R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM, 2013 Proceedings IEEE*, pp. 2130–2138, April 2013.

[26] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 1032–1041 vol.2, 2002.

[27] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control." RFC 5681 (Draft Standard), Sept. 2009.

[28] R. Dimond, E. Blanton, and M. Allman, "Practices for TCP Senders in the Face of Segment Reordering,"

[29] K.-C. Leung, V. Li, and D. Yang, "An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, pp. 522–535, April 2007.

[30] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "Rr-tcp: a reordering-robust tcp with dsack," in *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pp. 95–106, Nov 2003.

[31] K.-C. Leung and C. Ma, "Enhancing tcp performance to persistent packet reordering," *Communications and Networks, Journal of*, vol. 7, pp. 385–393, Sept 2005.

[32] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka, "A new tcp for persistent packet reordering," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 369–382, Apr. 2006.

[33] S. Bhandarkar, N. Sadry, A. Reddy, and N. Vaidya, "Tcp-dcr: a novel protocol for tolerating wireless channel errors," *Mobile Computing, IEEE Transactions on*, vol. 4, pp. 517–529, Sept 2005.

[34] R. Ludwig and A. Gurtov, "The Eifel Response Algorithm for TCP." RFC 4015 (Proposed Standard), Feb. 2005.

[35] R. Ludwig and M. Meyer, "The Eifel Detection Algorithm for TCP." RFC 3522 (Experimental), Apr. 2003.

[36] J. Nagle, "Congestion Control in IP/TCP Internetworks." RFC 896, Jan. 1984.

[37] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit." RFC 3042 (Proposed Standard), Jan. 2001.

[38] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith, "Privacy Considerations for Internet Protocols." RFC 6973 (Informational), July 2013.

[39] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 64–74, July 2008.

[40] S. Liu, T. Basar, and R. Srikant, "Tcp-illinois: A loss- and delay-based congestion control algorithm for high-speed networks," *Performance Evaluation*, vol. 65, no. 6-7, pp. 417 – 440, 2008. Innovative Performance Evaluation Methodologies and Tools: Selected Papers from ValueTools 2006.

[41] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control." RFC 2581 (Proposed Standard), Apr. 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[42] E. Blanton, M. Allman, K. Fall, and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP." RFC 3517 (Proposed Standard), Apr. 2003. Obsoleted by RFC 6675.

[43] M. Mathis, N. Dukkipati, and Y. Cheng, "Proportional Rate Reduction for TCP." RFC 6937 (Experimental), May 2013.

[44] L. A. D. Hayes, D. Ros and S. Floyd, "Common TCP Evaluation Suite," July 2014.

[45] S. Hemminger, "iproute2." `http://git.kernel.org/cgit/linux/kernel/git/shemminger/iproute2.git`, 2014. [Online; accessed Dec-2014].

[46] M. A. Brown, "Traffic Control HOWTO." `http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/overview.html`, 2006. [Online; accessed Dec-2014].

[47] B. Hubert, "Linux Advanced Routing and Traffic Control HOWTO." `http://tldp.org/HOWTO/Adv-Routing-HOWTO/index.html`, 2002. [Online; accessed Dec-2014].

[48] D. C. Martin Devera, "HTB Linux queuing discipline manual - user guide." `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`, 2002. [Online; accessed Dec-2014].

[49] T. Boot, "[Netem] Jitter without packet reordering." `https://lists.linux-foundation.org/pipermail/netem/2013-August/001582.html`, 2013. [Online; accessed Dec-2014].

[50] reqres, "cdma2000 Evaluation Methodology Revision 0," tech. rep., 3GPP2, 2004.

[51] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm." RFC 6582 (Proposed Standard), Apr. 2012.

[52] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 64–74, July 2008.

[53] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 92–99, Jan. 2010.

[54] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10', (New York, NY, USA), pp. 267–280, ACM, 2010.

[55] E. Dumazet, "kernel/git/torvalds/linux.git - Linux kernel source tree."

[56] "sysctl(8) - Linux man page."