CHALMERS

Analysing TCP performance when link experiencing packet loss

Master of Science Thesis [in the Programme Networks and Distributed System]

SHAHRIN CHOWDHURY
KANIZ FATEMA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2013

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Analysing TCP performance when link experiencing packet loss

SHAHRIN CHOWDHURY,
KANIZ FATEMA

© SHAHRIN CHOWDHURY, October 2013.
© KANIZ FATEMA, October 2013.

Examiner: TOMAS OLOVSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2013

Acknowledgement

We are grateful to our supervisor and examiner Tomas Olovsson for his valuable time and assistance in compilation for this thesis. We would like to pay our gratitude to CHALMERS for providing us necessary equipment. We would also like to thank the Department of Networks and Distributed Systems for giving us the opportunity to perform our thesis in the department. Last but not the least we are grateful to our parents for encouraging us to achieve our goal.

Abstract

TCP is a reliable protocol which is capable of handling retransmission and packet loss. In TCP, packet loss is not expected to have a noticeable impact on bandwidth. However, performance was affected even at low packet loss rates (1%) and with an increased rate of packet loss, a drastic drop was observed. To uncover the cause of this unexpected behavior of TCP, a deep analysis of TCP has been accomplished.

In this paper we have done a comparison between three different congestion control algorithms (Cubic, Reno and H_TCP) and a deeper analysis of Reno by means of several experimental tests in when the link experiences Data loss and ACK loss. Initially different TCP congestion control algorithms were used to observe their influence on bandwidth rate. Subsequently the TCP variables i.e. advanced window scaling, ECN value, window scaling, TCP no-metric-save value were changed to examine their role in obtaining adequate bandwidth rate with respect to packet drop. In addition to our experimental results, we also include some possible reasons behind the drastic drop in performance rate which was observed. Moreover, experimental results show that the congestion control algorithm H-TCP performed better than Cubic and Reno while link was experiencing packet loss. However, ACK loss didn't affect performance that much, and up to 50% loss of ACKs could be tolerated with almost no performance degradation.

Keywords: TCP congestion control algorithms, TCP variables, Data loss, ACK loss, Packet loss, Bandwidth rate etc.

Table of Content

Chapter 1: Introduction 12

1.1 Background........ 12

1.2 Outline of the paper 12

Chapter 2: Problem description 13

2.1 Detected problem in TCP performance........ 13

2.2 Related work 14

Chapter 3: TCP Concepts 17

3.1 TCP congestion avoidance algorithms 17

3.2 TCP variables 18

Chapter 4: Testing TCP performance 20

4.1 Description of measurement tools 20

4.1.1 Wireshark........ 20

4.1.2 Linux Network Traffic control (tc) 20

4.1.3 IPERF 21

4.1.4 Netcat 21

4.2 Setting up the network........ 22

4.3 Measuring the impact of TCP drops........ 24

4.3.1 Experiment when Data link experiencing Data Drop 24

4.3.2 Experiment when Data link experiencing ACK drop........ 25

4.3.3 Tests with CUBIC when experiencing Data Drop 26

4.3.4 Tests with CUBIC when experiencing ACK drop 27

4.3.5 Tests with Reno when experiencing Data drop 28

4.3.6 Tests with Reno when experiencing ACK drop 28

Chapter 5: Tests with more algorithms and TCP variables 30

5.1 Experiments with different congestion avoidance algorithms for data loss 30

5.1.1 CUBIC 31

5.1.2 RENO 31

5.1.3 H-TCP 32

5.1.4 Tests with different TCP variables: tcp_window_scaling 33

5.1.5 Tests with different TCP variables: tcp_adv_win_scale 34

5.1.6 Tests with different TCP variables: tcp_ecn 35

5.1.7 Tests with different TCP variables: tcp_no_metrics_save value 35

5.2 Experiments with different congestion avoidance algorithms for ACK loss............36

5.2.1 CUBIC/RENO/H_TCP 36

5.2.2 Tests with different TCP variables: tcp_window_scaling 37

5.2.3 Tests with different TCP variables: tcp_adv_win_scale 38

5.2.4 Tests with different TCP variables: tcp_ecn 39

5.2.5 Tests with different TCP variables: tcp_no_metric_save value 39

Chapter 6: Discussion 41

6.1 Outcome from the experiments when link experiencing Data loss 41

6.2 Outcome from the experiments when link experiencing ACK loss........42

6.3 Possible reason behind the drastic drop when link experiencing Data loss in RENO 45

Chapter 7: Conclusion 47

Chapter 8: Future Work 49

References 50

Appendix 52

1. Tests with different TCP variables in CUBIC for Data loss 52

1.2 tcp_adv_win_scale 52

1.3 tcp_ecn 53

1.4 tcp_no_metric_save 53

2 Tests with different TCP variables in CUBIC for ACK loss 54

2.1 tcp_window_scaling 54

2.2 tcp_adv_win_scale 54

2.3 tcp_ecn 55

2.4 tcp_ no_metric_save 55

Table of Figures

Figure 1: Throughput for Data drop (SSH over UDP) 13

Figure 2: Throughput for ACK drop (SSH over UDP) 14

Figure 3: IP Packet handling in the Linux Kernel 21

Figure 4: Network Topology 22

Figure 5: Test Network (Before modifying the NIC) 23

Figure 6: Simple network 24

Figure 7: Performance rate of data drop 25

Figure 8: Performance of ACK drop 25

Figure 9: Performance for Data drop in CUBIC 27

Figure 10: Performance for ACK drop in CUBIC 27

Figure 11: Performance for Data drop in Reno 28

Figure 12: Performance for ACK drop in Reno 28

Figure 13: Performance for Cubic 31

Figure 14: Performance for Reno 32

Figure 15: Performance for H-TCP 32

Figure 16: Performance of tcp_window_scaling when turned on 34

Figure 17: Performance of tcp_adv_win_scale when turned on 34

Figure 18: Performance of tcp_ecn value when turned on 35

Figure 19: Performance of tcp_no_metric_save value when turned on 36

Figure 20: Performance for ACK loss with all three congestion avoidance algorithm (Cubic, Reno and H-TCP) 36

Figure 21: Performance for ACK loss with Reno when TCP variables were turned off 37

Figure 22: Performance of tcp_window_scaling when turned on 38

Figure 23: Performance of tcp_adv_win_scale when turned on 38

Figure 24: Performance of tcp_ecn when turned on 39

Figure 25: Performance of tcp no_metric_save value when turned on 39

Figure 26: Performance of three different congestion avoidance algorithms while experiencing data loss 41

Figure 27: Comparison between four different TCP variables when turned off and when set to their default values while experiencing data loss 42

Figure 28: Comparison of the three different congestion avoidance algorithms in case of ACK loss .. 43

Figure 29: Performance of TCP Congestion Avoidance algorithms while experiencing different amount of ACK loss 43

Figure 30: Comparison between four different TCP variables when turned off and TCP variables when set to their default values while experiencing ACK loss 44

Figure 31: Performance during congestion in the general network 45

Figure 32: Drastic drop in performance in the experimental network 46

Figure 33: Performance of tcp_window_scaling when turned on 52

Figure 34: Performance for tcp_adv_win_scale when turned on 52

Figure 35: Performance of tcp_ecn when turned on 53

Figure 36: Performance of tcp_no_metric_save when turned on 53

Figure 37: Performance of tcp_window_scaling when turned on 54

Figure 38: Performance of tcp_adv_win_scale when turned on 54

Figure 39: Performance of tcp_ecn when turned on 55

Figure 40: Performance of tcp_no_metric_save when turned on 55

Chapter 1: Introduction

1.1 Background

TCP is a protocol offering reliable communication between two communicating parties. It takes care of communication problems from the underlying layers, it retransmits lost packets (segments), detects duplicates and reorders unordered packets when needed, problems that may be caused by network congestion, dynamic load balancing and other unpredictable events.

TCP uses a number of mechanisms to deal with network congestion and packet loss. These mechanisms control the rate of traffic in order to maximize performance and network link utilization. When packet loss increases in the network, the number of retransmissions increase which, in turn, decreases effective bandwidth and causes longer and more unpredictable delays.

In this thesis work, we have analyzed TCP behavior when it experiences packet drops in a small network. We have found that TCP is unable to handle even 1% packet loss. We would have expected a reasonable small performance drop, but instead a drastic drop in performance was observed. We have investigated the nature of this problem and tested different TCP congestion avoidance algorithms under Linux (e.g. Reno, Cubic and H-TCP). We have also investigated the effect of changing some TCP parameter settings, such as tcp_adv_win_scale, tcp_ecn, tcp_window_scaling, tcp_no_metrics_save.

1.2 Outline of the paper

The outline of this paper is as follows. First, Chapter 2 explains the detected problem of TCP performance, and related work. In Chapter 3 contains a TCP concepts of the various TCP congestion avoidance algorithms and TCP variables have been described briefly. Chapter 4 continues with a description of measurement tools used in this thesis work, basic setup of the network along with test scenarios and measurements of packet drop has presented. Chapter 5 contains an analysis of all experiments performed. In Chapter 6 the paper is summarized, Chapter 7 contains conclusions and finally Chapter 8 discusses future work.

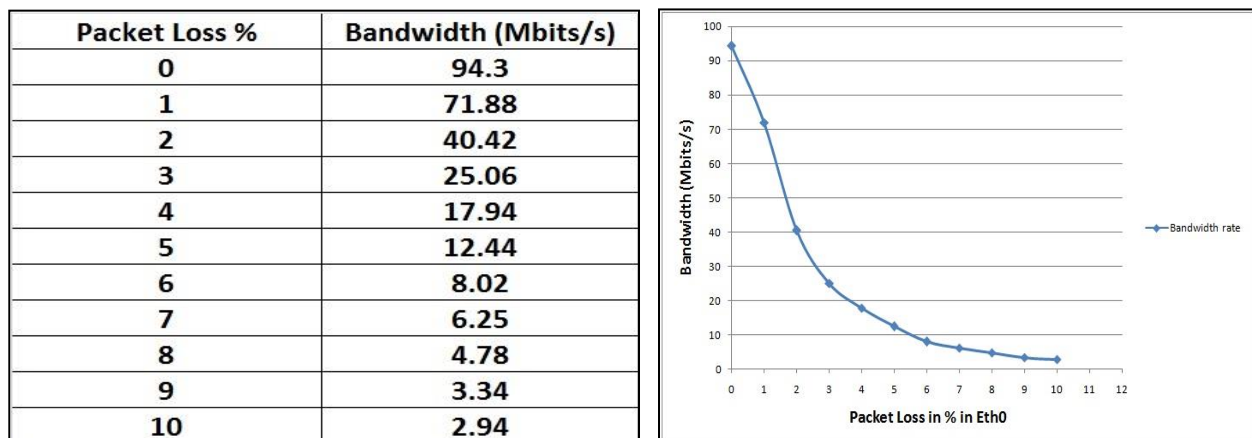
Chapter 2: Problem description

2.1 Detected problem in TCP performance

This thesis is an extension of a previous thesis called 'SSH Over UDP': "In 'SSH over UDP' a UDP based solution was implemented for OpenSSH. OpenSSH uses tunneling of one TCP connection within another TCP connection for its VPN features. But various sources claim that using such TCP in TCP in practice; will raise some conflicts between the two TCP implementations especially when experiencing packet loss. The general recommended is therefore to avoid tunneling TCP in TCP. The goal with their work was to see whether a UDP based solution works better than a TCP based solution when links experience packet loss." [4]

Tunneling is a secure and trusted way to transfer data between the two networks. Tunneling protocols may use data encryption to transport the data over a public network. Details information about Tunneling, OpenSSH and VPN is not included in this thesis.

While they were testing the throughput for different rates of packet loss, a problem was detected. The throughput had dropped more drastically than expected when packet loss increased. Finding out the reason behind the unexpected performance drop; therefore become the scope of our research project.
The identified problem is shown in Figure 1 below.



Packet Loss %	Bandwidth (Mbits/s)
0	94.3
1	71.88
2	40.42
3	25.06
4	17.94
5	12.44
6	8.02
7	6.25
8	4.78
9	3.34
10	2.94

Figure 1: Throughput for Data drop (SSH over UDP)

When no packet loss was experienced in the network the throughput was 94.3 Mbits/s. With only 3% packet drop, throughput decreased to 25.06 Mbits/s, thus a huge drop in performance was observed. We expected that TCP would have adequate functionality to handle packet

drops in the network, at least when the link only experiences 3% packet drop. Even in the case of 1% packet loss, performance dropped from 94.3 Mbits/s to 71.88 Mbits/s which is also noticeable to the user.

Packet Loss %	Bandwidth (Mbits/s)
0	94.26
1	94.34
2	94.3
3	94.3
4	94.34
5	94.3
6	94.32
7	94.32
8	94.38
9	94.34
10	94.32

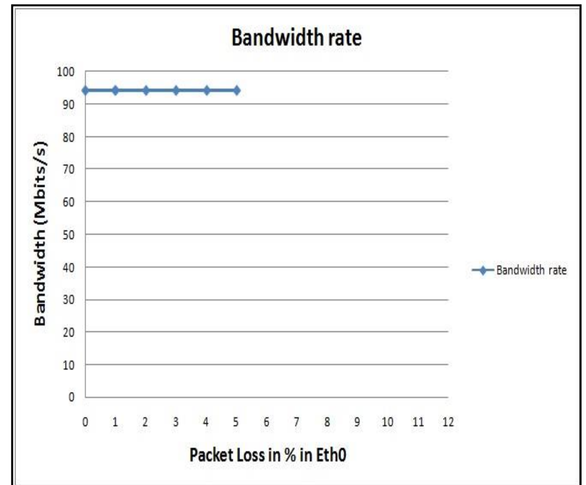


Figure 2: Throughput for ACK drop (SSH over UDP)

On the other hand, from Figure-2 it is possible to see that there is almost no variation in the throughput when ACKs were dropped.

TCP shows different behavior for data drop and ACK drop and the reasons behind this dissimilar behavior will be discussed later in this thesis.

2.2 Related work

Robert Morris [13] has described in the paper, the effect of congestion window size on TCP flow control. The author has identified that TCP's minimum congestion window of one packet is the reason behind the reduced data flow which occurs due to packet loss. Packet transmission rate decreases up to 50% after a single packet loss in the network which in turns generates delays which will be noticeable to the users. Routers also drops packet when their queue is overflowed. TCP is able to overcome the loss of a packet in a single round trip time by increasing its window size larger than 4 packets. The author has suggested a solution by increasing router buffer space together with limits on per-flow queue length. In this paper no external packet loss has been introduced; but for a single packet loss up to 50% decrease in

performance has been observed. Hence, initiating external packet loss will end up with huge performance loss.

Christian *et al.* [14] has offered a complete comparison among TCP's 12 different variants i.e. TCP (TCP Reno) and its improved version (TCP Vegas), variants for wireless networks (TCP Veno and TCP Westwood), variants for high-speed networks (TCP BIC, TCP CUBIC), and variants satellite networks (HSTCP, TCP Hybla, TCP Illinois, Scalable TCP, and TCP YeAH), and also a low-priority version (TCP-LP) in terms of achieved throughput and fairness. TCP YeAH performed a little bit better compared to Scalable TCP and TCP Illinois, considering achieved throughput. On the other hand when comparing fairness to other TCP variants; TCP-LP showed much lower throughput. From this paper, we have taken Reno and Cubic throughput graphs and compared them with the graphs generated in our experiments.

Honda and Osamu [2] presented in their paper that, end-to-end TCP performance is thought to be affected by some factors such as link bandwidth, propagation delay, MTU and router buffer size. The authors have focused on how these variables as well as the TCP tunneling affect end to end performance. From the observation it has been found that TCP tunneling causes degradation in end-to-end goodput when the network propagation delay is large. The buffer size has impact on goodput regardless of whether it's for end host or tunnel connections, which means that the decreasing goodput can be analyzed with the use of small buffer size. The paper has also proposed that the SACK option can be an alternative to increase goodput.

Titz [3] stated that TCP over TCP is thought to be an inappropriate combination when tunneling TCP over TCP due to the different timers in the different TCP stacks, which results in a meltdown problem. Due to the presence of a faster timer in the upper layer of TCP, it performs more retransmissions compared to the lower timer, which means that stacking of TCP messages makes the connection to break down as supposed to plain TCP which is able to bear more packet loss as it can avoid congestion. Because of these problems the author has suggested to use UDP as an alternative of TCP for the SSH tunnel.

Canberk, Berk and Jaya Dhanesh [5] have focused on the TCP meltdown problem. The hypothesis was that using the same TCP implementation for the tunnel connection and the

forwarded connection leads to TCP meltdown. More over using same TCP implementation one over the other, increases the performance loss almost twice compared to a single TCP connection. To avoid this, the authors suggest the use of mixed TCP implementations, for example a TAHOE implementation of TCP for forwarded connection and a Reno implementation for the other. Another suggestion was to use UDP instead of TCP when tunneling traffic. The authors concluded with the statement that TCP over TCP is not as bad as it thought to be; moreover using UDP creates new firewall problems.

Lee *et al.* [6] present in their paper a comparison between TCP and UDP traffic in order to recognize which one performs the best in the TCP tunnel when there is a bottleneck in the link. Only the path MTU size was investigated to analyze the performance of the TCP flow, and factors like link bandwidth, buffer size, propagation delay was ignored. In addition, other things like the number of TCP flows, TCP patterns and TCP tunnels, the traffic pattern of TCP flows, TCP congestion avoidance algorithms, presence of SACK option and the socket buffer size was also not considered.

Chapter 3: TCP Concepts

In this chapter we give a brief overview of various TCP congestion avoidance algorithms and different TCP variables analyzed later in this thesis work.

3.1 TCP congestion avoidance algorithms

"The TCP Congestion avoidance algorithm is the primary basis for congestion control in the internet"[7]. The congestion avoidance algorithm detects congestion by observing retransmission timer expiration and the reception of duplicate ACKs. As a remedy to this situation, the sender decreases its transmission window, i.e. its number of unacknowledged packets in transit to one half of the current window size.

There are various types of congestion avoidance algorithms. For example: Tahoe, Reno, New Reno, Vegas, BIC, CUBIC, H-TCP etc. In this work, three algorithms were chosen for analysis: CUBIC, Reno and H-TCP.

When we installed the Linux operating system, the default TCP congestion avoidance algorithm was Cubic. Cubic implements an enhanced congestion control algorithm called Binary Congestion Control shortly BIC. The BIC congestion window control function improves the RTT (round trip time) fairness and TCP friendliness. The BIC growth function plays an aggressive role for TCP in low speed networks with short RTTs. Cubic is compatible with both short and long RTTs which mean good scalability. "The congestion epoch period (time period between two consecutive loss events) of CUBIC is determined by the packet loss rate alone."
The main feature of CUBIC is that its window growth function is defined in real-time so that its growth will be independent of RTT [8].

In TCP, normally for each received ACK, the congestion window is increased by one segment per round trip time (RTT) this mechanism is called slow start. On the other hand when packet loss occurs; TCP applies a mechanism called Multiplicative Decrease Congestion Avoidance (MDCA) which decreases congestion window to half per round-trip time [24].

Timeout occurs when sender doesn't receive ACK within a given time. This initiates retransmission of lost segment. Fortunately, Reno has got fast retransmission feature which

reduces the time a sender waits before retransmitting. But MDCA mechanism doesn't work properly in Reno; so the fast retransmitted packets also starts dropping at the receiver end. Because retransmission rate is not proportionate to the receivers receiving capability which results in huge number of packet drops. Hence, Reno only performs well against very small packet loss [8,24].

Retransmission rate is not proportionate to the receivers receiving capability which results in huge number of packet drops.

H-TCP is the third congestion avoidance algorithm used in our experiments. H-TCP has some strong features; it can operate in a high speed network and H-TCP also performs better in long-distance networks. This protocol is capable of rapidly adapting to changes in existing bandwidth, which makes it a bandwidth efficient protocol [15].

3.2 TCP variables

A number of TCP variables were assumed to have influence on performance. Because of the different features the TCP variables control, we believed that they should have some impact on performance. In order to verify it, these four TCP variables were analyzed in this work.

tcp_window_scaling enables a TCP option which makes it possible to scale TCP windows to a size to accommodate "Large Fat Pipes (LPF)" i.e. links with a high bandwidth-delay product.TCP may experience bandwidth loss while passing TCP packets over the large pipes; as the channels are not being fully filled while waiting for ACK's for previously transmitted segments. The TCP window scaling option allows the TCP protocol to use a scaling factor to the windows, i.e. to use window sizes larger than the original TCP standard defined and ensures utilizing nearly all of the available bandwidth. The tcp_window_scaling is by default is set to 1, or true. To turning it off set it to 0[9, 16]. We have analyzed this feature in order to reduce performance loss while passing large amounts of data (10 MB, 25 MB etc) through the network.

The total buffer space available to a socket is shared between a buffer for the incoming data (TCP receiver window) and an application data buffer. The **tcp_adv_win_scale** is used to inform the kernel about how much socket buffer size should be saved for the application buffer and how much should be used for the TCP window size. The tcp_adv_win_scale is by default set to 2 which mean that the application buffer is using one fourth of the total space [9, 16]. In our experiments initially it was set to 0 which means that all available memory is allocated to the incoming data buffer. Afterwards we have checked its impact on performance by setting it to default value. We used this variable with the expectation that it's going to reduce performance loss more than what the tcp_window_scaling did.

The **tcp_ecn (Explicit Congestion Notification)** variable automatically informs the host when there is congestion in a route towards a specific host or a network by turning on an Explicit Congestion Notification in TCP connections. The tcp_ecn by default is turned on and set to 2, on the other hand to turn it off change the value to 0. To turn it on in the kernel it should be set to 1[9, 16]. This TCP variable was chosen to check whether there is any congestion in the network.

The **tcp_no_metrics_save** controls a function that allows the system to remember the last slow start threshold (ssthresh) when it is turned on. This feature modifies the result of the second test and leads to error. When the previous connection holds the better result the outcome of the second test will show a modified result (better) rather than the original result. The tcp_no_metrics_save in default is set to 0 and to turn this function off, it is set to1 [9, 16]. The reason behind selecting this variable was to observe its impact on performance, as it normally offers an overall performance improvement since by remembering what characteristics the link had last time, it does not have to start so slowly when searching for the congestion point.

Chapter 4: Testing TCP performance

This chapter contains a description of the test network including the network configuration and also about the data obtained from the tests. Our initial data is presented and is also compared with the test data achieved by the “SSH Over UDP” project.

4.1 Description of measurement tools

The following tools have been used when testing TCP performance, and a brief introduction of the tools are presented below.

4.1.1 Wireshark

Wireshark is a network protocol analyzer developed by an international team of networking experts. It is a tool for capturing traffic on a computer network and also one of the best open-source tools for displaying the contents from a network packet. It is the de facto standard across many industries and educational institutions. It runs on Windows, UNIX, OS X [10, 19, 20].

4.1.2 Linux Network Traffic control (tc)

Linux Network Traffic Control, tc, was the main network emulator in our tests. “It enables us to realistically recreate a wide variety of network conditions like packet loss/error/reordering, latency, bandwidth restrictions and jitter.”[22] Only unidirectional packet loss was emulated during the tests. Using the ‘tc’ command, it is possible to specify the percentage of random packet loss. For initializing packet loss on the router host the following command was used:

tc qdisc add eth<x> root netem loss <percentage> [4,17]

The procedure of handling an IP packet inside a Linux-based router is shown in Fig.4.When a packet enters a network interface card (NIC), it is classified (see below) and enqueued (in ingress qdisc) before going through linux internal packet handling. Upon the completion of

the packet handling procedure, packets are again classified and enqueued (in egress qdisc) for transmission on the egress NIC [23].

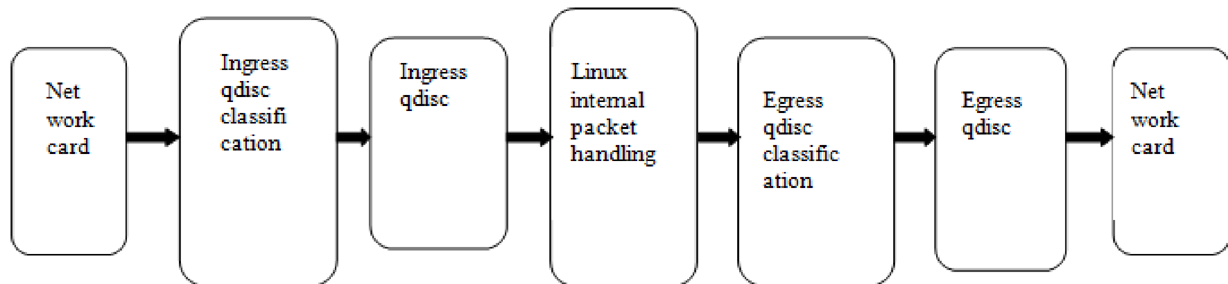


Figure 3: IP Packet handling in the Linux Kernel

Packet header fields (source and destination IP addresses, port numbers etc) are analyzed when performing packet classification and the Traffic control tool, tc, is used to configure qdisc (short for queuing discipline) as well as configure packet classification into qdisc. Packets are enqueued in one of the ingress/egress queues, based on this classification. A FIFO method is then applied to process packets. Qdisc is defined as, the combination from queue and algorithm that decides when to send which packet [23].

4.1.3 IPERF

IPERF was developed by NLANR/DAST and it's a network testing tool that is able to generate TCP and UDP data streams and measure TCP and UDP performance. IPERF reports bandwidth, delay jitter and datagram loss. Various parameters e.g. data size, testing time etc., can be set in IPERF in order to test the network. IPERF consists of a client a server and allows for unidirectional or bi-directional throughput measurements [11].

4.1.4 Netcat

Netcat is a simple networking utility for reading from and writing to network connections using TCP or UDP. It is a feature-rich network debugging and exploration tool; and it is capable of creating almost any kind of connection one would need and has several interesting built-in capabilities. This tool is designed in such a way that it can be run by other programs and scripts [12, 21].

In the beginning of the experiments both the IPERF and Netcat was used to generate the traffic. Since they have shown the same results, we selected IPERF for the rest of the experiments.

4.2 Setting up the network

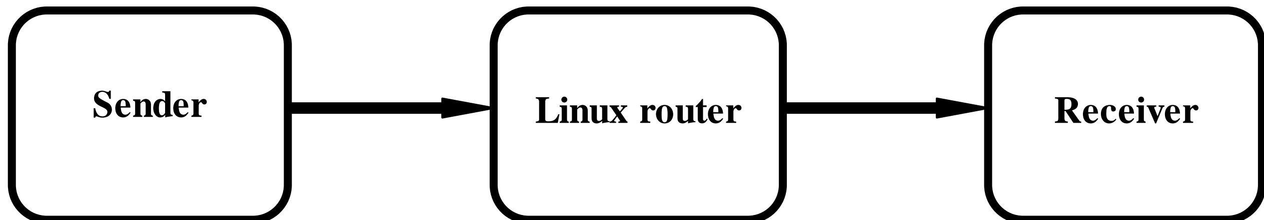


Figure 4: Network Topology

A small network was created consisting with two hosts (server and client) and a middle computer (a router). The middle computer was used to disturb the communication between the server and the client by introducing packet loss. Wireshark was used to analyze the traffic passing between the server and the client. Linux Network Traffic Control, tc was used to introduce packet loss in the router. We then used IPERF on the client host to generate traffic where the amount of data sent on the link was specified. The Maximum transmission unit (MTU) was also set to its default value which is used to specify the largest data unit that is allowed to be forward through a link. Ethernet at the link layer allows maximum 1500-byte packets. In this thesis work, the MTU size was set to 1400-bytes [18].

The configuration of Host A:

- ✓ CPU: AMD Athlon ® 64 Processor 3200+ 2000MHz
- ✓ Operating system: Ubuntu Server (Linux)
- ✓ Network card: 100 Mbits (Eth0)

The configuration of Host C:

- ✓ CPU: Intel® Pentium ® 4 3.60GHz
- ✓ Operating system: Ubuntu Client (Linux)
- ✓ Network card: 1000 Mbits (Eth2)

The configuration of Router:

- ✓ CPU: Intel® Pentium ® 4 3.00GHz
- ✓ Operating system: Debian (Linux)
- ✓ Network cards: 100 Mbit (Eth0) and 1000 Mbit (Eth2).

A TCP connection was established between the two end hosts. The Network Interface Card (NIC) was configured for 100Mbit/s on host A and 1000Mbit/s on host C. Later on it was modified to be same for all hosts.

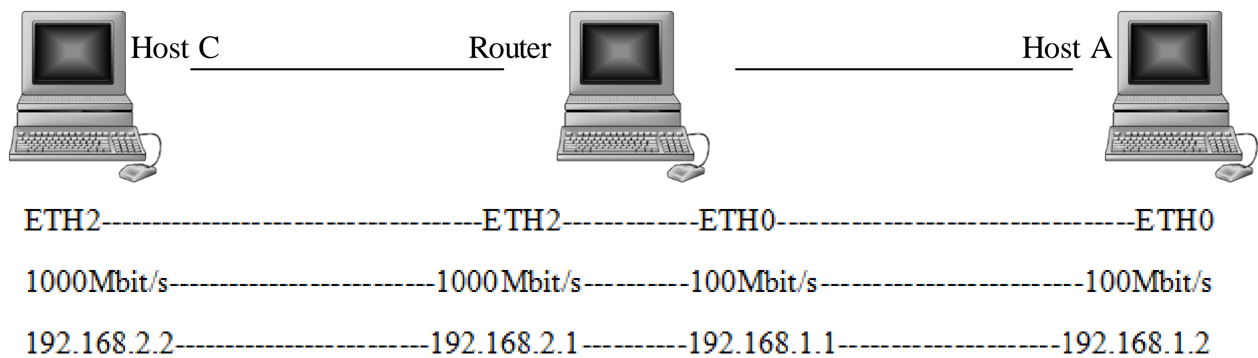
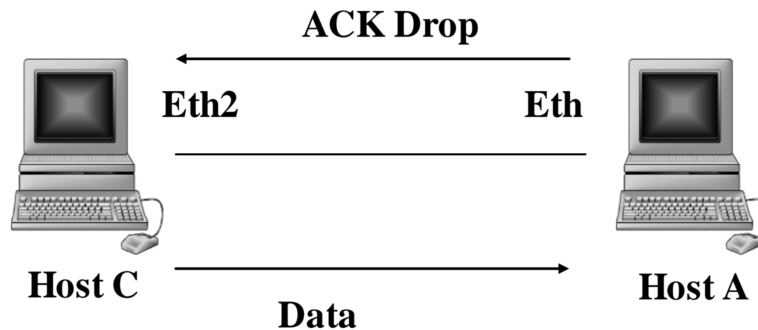


Figure 5: Test Network (Before modifying the NIC)

The above figure is the physical picture of the network. A more simplified picture is shown below in order to understand the test scenarios more clearly.



Measured bandwidth in Eth0 = Bandwidth rate for Data

Measured bandwidth in Eth2 = Bandwidth rate for ACK (Acknowledgement)

Figure 6: Simple network

4.3 Measuring the impact of TCP drops

Initially the default congestion avoidance algorithm in the Linux TCP implementation called Cubic (See Chapter 3.1) was running in the three computers. TCP variables were set to their default value (See Chapter 3.2). Using IPERF, performance was measured during 30 seconds. When packet loss increased, performance decreased. For 0% packet loss the performance was 94.1 Mbits/s whereas for 10% packet loss it decreased to 2.89 Mbits/s. The initial data and the graph for data drop are presented in the following sub sections.

Measurement specifications for the following experiments are:

Emulator: tc, bandwidth measurement: IPERF, test-time: 30 s

Link speed: 1000Mbits/s- Host C, 100Mbits/s – Host A

Maximum Transmission Unit (MTU), Router=1500, Host A&C=1400

4.3.1 Experiment when Data link experiencing Data Drop

In this first experiment, only six measurement values were taken to observe whether the network really experienced a drastic drop when the link experienced data loss. After this experiment our experimental results confirm the results obtained by the "SSH Over UDP" project [4].

Packet Loss %	Bandwidth (Mbits/s)
0	94.1
1	88.5
2	59.5
3	40.2
4	25.5
10	2.89

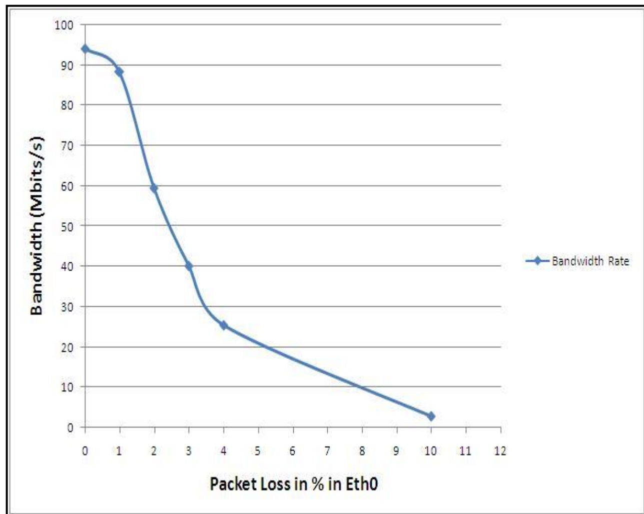


Figure 7: Performance rate of data drop

From the above figure (Figure 7) it can be clearly seen that performance decreased in a drastic way when data loss was introduced to the network. The bandwidth was almost fully used for data transmission when there were no data losses (or 0% packet loss) in the network. Surprisingly, even when introducing only 1% packet loss, the performance drops to 88.5 Mbits/s. And the decrease continues down to 2.89 Mbits/s when data loss reaches 10%.

4.3.2 Experiment when Data link experiencing ACK drop

Six measurement values were taken to observe the performance when ACKs were dropped instead of data. No performance drop was found as in the previous test (4.3.1).

Packet Loss %	Bandwidth (Mbits/s)
0	94.2
1	94.2
2	94.2
3	94.2
4	94.2
10	94.2

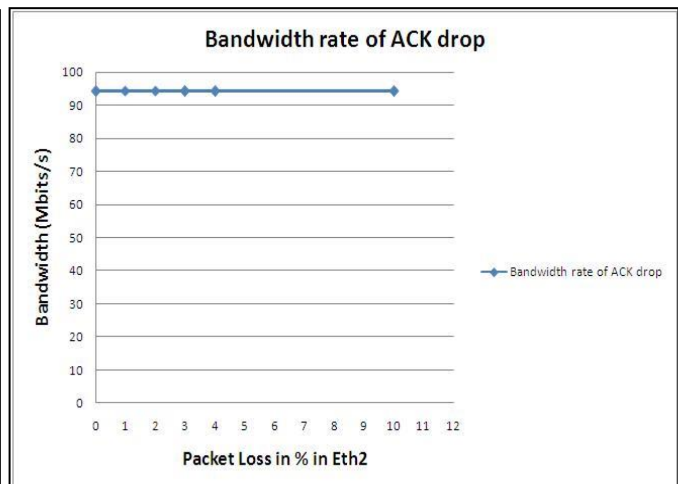


Figure 8: Performance of ACK drop

The described drastic drop when experiencing data loss mentioned in the "SSH Over UDP" [4] (Stated in chapter 2 section 2.1) was also experienced in this network. To verify this drastic drop in performance some more measurements were made. These results are shown in section 4.3.3, 4.3.4, 4.3.5 and 4.3.6. Some changes were made for these experiments. Instead of transmitting the data for 30 seconds, 25MByte of data was transferred. MTU was set to 1400 for host A, C and the Router.

4.3.3 Tests with CUBIC when experiencing Data Drop

For this experiment some changes were made. CUBIC was again chosen because it is the default congestion avoidance algorithm in the systems running on the two hosts and in the router. Only values that have changed from the previous experiments are given below.

Initialized Value:
Data Size =25 MB instead of test-time: 30 s
MTU = 1400 for all hosts

Using the above configuration; two tests were made. One for data drop and the other for ACK drop in the network.

While generating traffic for 30 seconds, IPERF was sending as much packets as it could within this period of time. WireShark was used in the router to inspect all these packets over the network. Processing and analyzing that amount of traffic was bit difficult for Wireshark which might had occurred because of the slower processing speed (router´s speed) of the CPU where wireshark was installed. As a result, Wireshark was taking more time to capture packets for every experiment. It was suspected that the large amount of data was creating an overflow in the data path which could be a possible reason behind the drastic drop. Hence, to simplify the analysis, a small amount of traffic (25MB) was generated.

Packet Loss %	Bandwidth (Mbits/s)
0	93.7
1	79.1
2	62.4
3	31.2
4	26.2
5	11.8
6	10.2
7	6.49
8	4.7
9	3.51
10	2.76

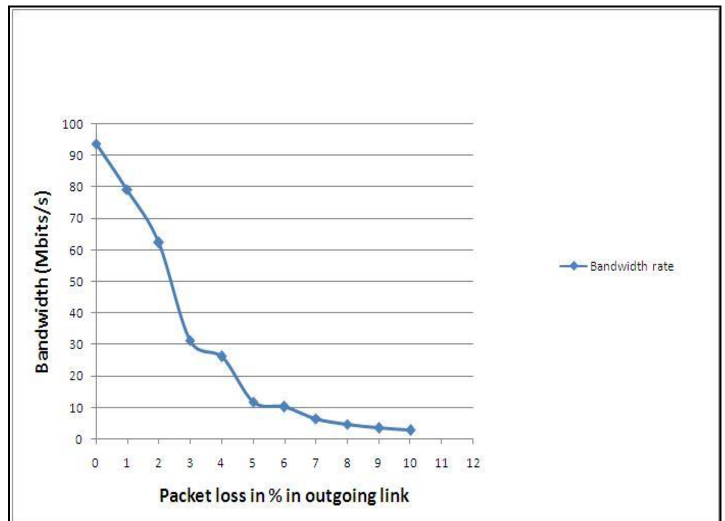


Figure 9: Performance for Data drop in CUBIC

The experienced performance when the link is experiencing data loss is shown in Figure 9. Here, again performance decreased radically with an increasing data loss.

4.3.4 Tests with CUBIC when experiencing ACK drop

A similar type of experiment in CUBIC was made for ACK drop. The obtained result is shown in the following Figure.

Packet Loss %	Bandwidth (Mbits/s)
0	93.7
1	93.7
2	93.7
3	93.7
4	93.7
5	93.7
6	93.7
7	93.7
8	93.7
9	93.7
10	93.7

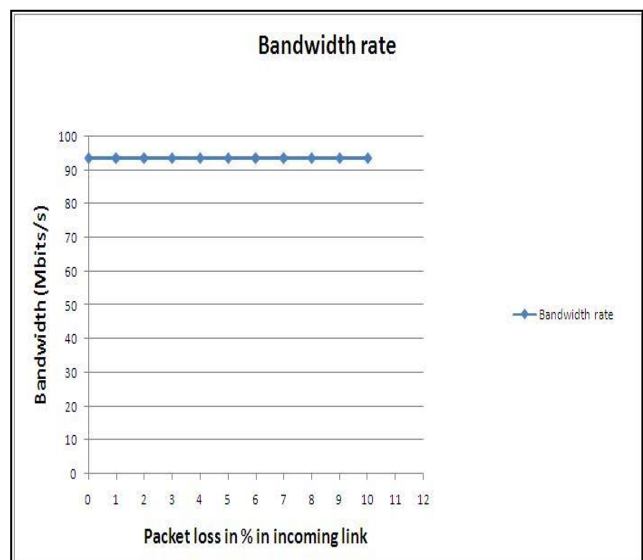


Figure 10: Performance for ACK drop in CUBIC

As shown in Figure 10, performance remains constant at 93.7 Mbits/s even when experiencing 10% of ACK loss.

4.3.5 Tests with Reno when experiencing Data drop

Similar experiments like 4.3.3 and 4.3.4 was now made with the Reno congestion avoidance algorithm (detailed information about Reno is given in chapter 3 section 3.1). The only change that was made here is the change of congestion avoidance algorithm. Data size was also kept to 25MB. Two tests were made with Reno and performance was measured both for data drop and for ACK drop. The tests outcomes are described below.

Packet Loss %	Bandwidth (Mbits/s)
0	93.7
1	93.6
2	71
3	54.1
4	24.9
5	10.4
6	8.87
7	5.51
8	3.46
9	2.64
10	1.93

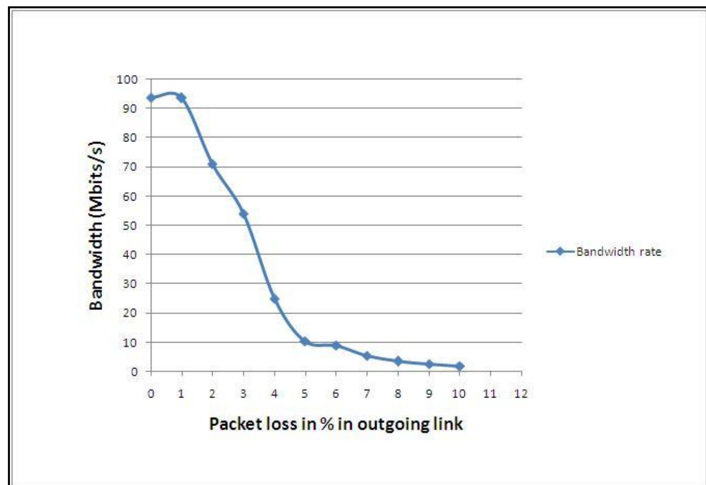


Figure 11: Performance for Data drop in Reno

From this analysis it was found that performance becomes more drastic than for CUBIC. The performance was the same when no data loss was introduced but for 10% data loss the rate falls into 1.93 Mbits/s. This rate is lower than the achieved performance in CUBIC (See Figure 8).

4.3.6 Tests with Reno when experiencing ACK drop

Packet Loss %	Bandwidth (Mbits/s)
0	93.7
1	93.7
2	93.7
3	93.7
4	93.7
5	93.7
6	93.7
7	93.7
8	93.7
9	93.7
10	93.7

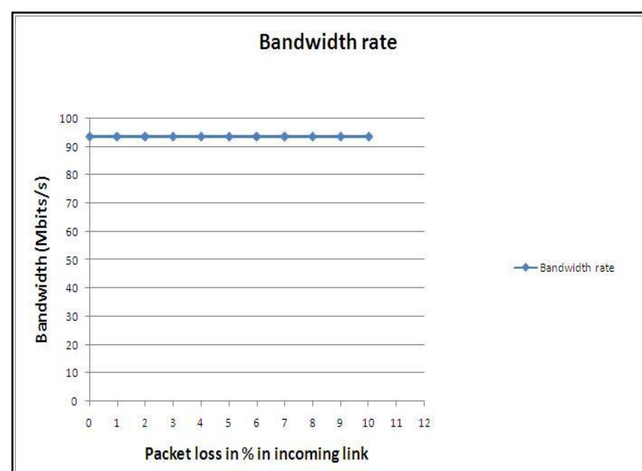


Figure 12: Performance for ACK drop in Reno

The above figure shows that the ACK drop in the network does not vary when the TCP congestion avoidance algorithm was changed to Reno. The performance remains same even for 10% packet loss 93.7 Mbits/s.

Chapter 5: Tests with more algorithms and TCP variables

To identify the reason behind TCP´s behavior as explained in chapter 4; several experiments were done both for links experiencing data drop and ACK drop. We also saw that different TCP congestion avoidance algorithms had some impact on performance while experiencing data drop, and that performance didn´t change for ACK drop. So, in order to observe the TCP performance during data loss; different congestion avoidance algorithms were tested i.e. Cubic, Reno and H-TCP along with changing four different TCP variables. The reason for experimenting with different TCP variables and their impact has been discussed in 5.1 and 5.2.

Before starting the experiments we have configured and changed some network settings. In the previous experiments, the speed of the NIC (Network Interface Card) for all three computers was not same, but it was set to the maximum value supported by the NIC´s. Hence to reduce the possible impacts on the experimental results, we now use the same values for all NICs.

5.1 Experiments with different congestion avoidance algorithms for data loss

Prior to start, all NICs are set to 10Mbit/s for all links. The link speed was reduced (from 100 Mbit/s to 10 Mbit/s on the server side and 1000 Mbit/s to 10 Mbit/s on the client side) to have the same link speed in the network. The same congestion avoidance algorithm was then set in all the three hosts. First, the hosts were configured with Cubic, and then changed to Reno and at last to H-TCP to observe the results. To begin with, all four TCP variables considered in this thesis (described earlier in chapter 3) were turned off.

All important settings for the experiments are highlighted below.

Tools used to analyze the data: IPERF

Link speed, in all links (outgoing and Incoming) = 10 Mbit/s

Data Size = 25MB

MTU for all three hosts = 1400

5.1.1 CUBIC

Cubic is the default congestion avoidance algorithm in the Linux operating system. So the first experiment was done with Cubic to observe the performance. From theory (details in chapter 3 sections 3.1) it is already known that Cubic is compatible with both short and long round trip times. It is also supposed to give good performance while links are experiencing data loss.

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.19
2	9.08
3	8.53
4	7.85
5	7.07
6	6.21
7	4.37
8	3.24
9	2.53
10	2.34

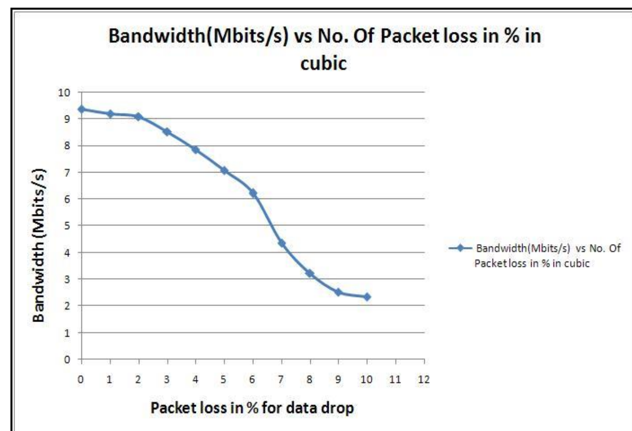


Figure 13: Performance for Cubic

From the above figure (figure 13) it can be seen that when the link was not experiencing any data loss, then the link was fully utilizing its bandwidth and the performance rate was quite high 9.37 Mbits/s. 1% of data loss decreased the performance rate to 9.19 Mbits/s. At 4% data loss, performance went down to 7.85 Mbits/s. After that, the decrease continued and for 10% of data loss performance was down to 2.34 Mbits/s.

5.1.2 RENO

From the earlier theoretical discussion (section 3.1), Reno was not expected to be a competitive congestion avoidance algorithm when experiencing large amount of data loss. Due to its fast retransmission policy, it worked well while small data loss was experienced.

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.18
2	9.13
3	8.38
4	7.14
5	5.86
6	4.64
7	3.95
8	2.87
9	2.22
10	1.93

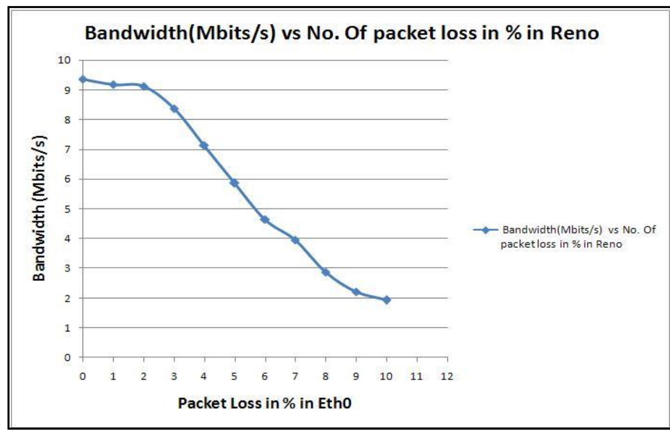


Figure 14: Performance for Reno

From figure 14 it can be seen that the initial performance was similar to Cubic for no data loss, and Reno also gave 9.37 Mbits/s in performance. But performance decreased drastically when experiencing 3% of data loss. Thereafter, performance went down with increased data loss. At 10% data loss, performance had decreased to 1.93 Mbits/s. The conclusion is that Reno is not able to handle data loss of 10%. It also performed worse than Cubic as was expected from the theory.

5.1.3 H-TCP

According to the theory (chapter 3.1) H-TCP has some strong features to handle data loss over a long distance network. It has also the capability of rapidly adapting changes in existing bandwidth which makes it a bandwidth efficient protocol. Because of these features, H-TCP was one of the selected algorithms among all the congestion avoidance algorithms for this work.

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.19
2	9.08
3	8.94
4	8.55
5	7.58
6	6.15
7	5.03
8	3.8
9	3.61
10	2.64

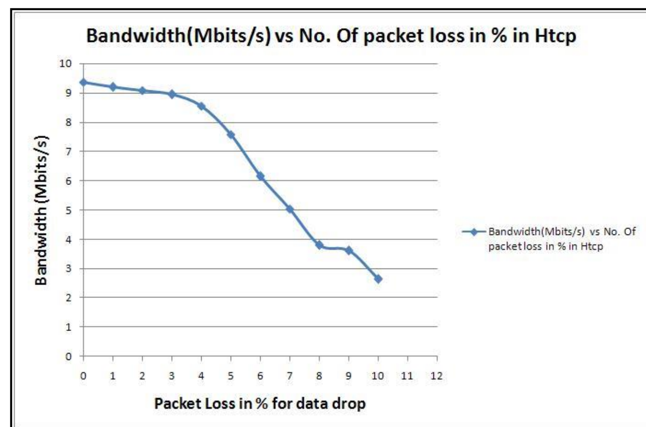


Figure 15: Performance for H-TCP

Figure 15 shows the performance of H-TCP. At the beginning when no data loss was introduced to the network, then H-TCP was also utilizing the full bandwidth of the link like

Cubic and Reno; and showed a high performance of 9.37 Mbits/s. But later on with an increasing rate of data loss in the network, it also experiences a performance drop. But this drop is slightly better than Cubic and Reno, and from the graph it can be seen that a more drastic drop starts for H-TCP from 5% of data loss. After introducing 10% data loss performance decreases to 2.64 Mbits/s.

From these experiments, we can conclude that different congestion avoidance algorithms have impact on performance when links experience data loss.

Although the performance of H-TCP was better than Reno and Cubic, for the further experiments with TCP variables Reno was chosen since it had the worst performance among these three congestion avoidance algorithms. It was expected that, if Reno manages to survive with the performance loss; the other two algorithms will definitely show better performance.

5.1.4 Tests with different TCP variables: tcp_window_scaling

A brief description and possible impact of the TCP variables has been described in section 3.2.

Initially, tcp_window_scaling was set to 1 or true (which means it was turned on). This means that it will automatically scale the window size according to large fat pipes which ensures utilizing nearly all available bandwidth, which directly reduces the performance loss. Tcp_window_scaling was turned off in the other experiments where the other TCP variables were changed. The network settings for this test are shown below.

Congestion Avoidance Algorithm = RENO

Tools used to analyze the data: IPERF

Link speed, in all links (outgoing and Incoming) = 10 Mbits/s

Data Size = 25MB

MTU for all three hosts = 1400

tcp_window_scaling = 1

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.28
2	9.02
3	8.4
4	6.75
5	5.62
6	4.57
7	3.71
8	2.73
9	2.31
10	1.91

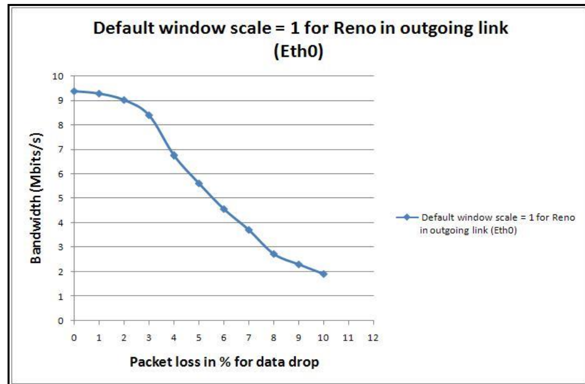


Figure 16: Performance of tcp_window_scaling when turned on

After setting the variable tcp_window_scaling = 1 the above table and graph was obtained. Even now with an increased amount of data loss, performance starts decreasing. For 3% of data loss, performance decreases to 8.4 Mbits/s, and for 10% data drop performance is down to 1.91 Mbits/s. Although this TCP variable has the function to reduce performance loss, in this case it doesn´t show any remarkable changes compared to figure 14 where tcp_window_scaling is turned off.

5.1.5 Tests with different TCP variables: tcp_adv_win_scale

In figure 14 tcp_adv_win_scale was turned off. Tcp_adv_win_scale was initially turned on and set to its default value 2, which means that the application buffer was using one fourth of the total space (details given in section 3.2). This TCP variable can be turned off by setting its value to 0. So, to observe the impact on performance of tcp_adv_win_scale, it was turned on and the other TCP variables were turned off.

tcp_adv_win_scale= 2

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.26
2	8.98
3	8.52
4	7.62
5	6.13
6	4.18
7	3.86
8	2.96
9	2.23
10	1.77

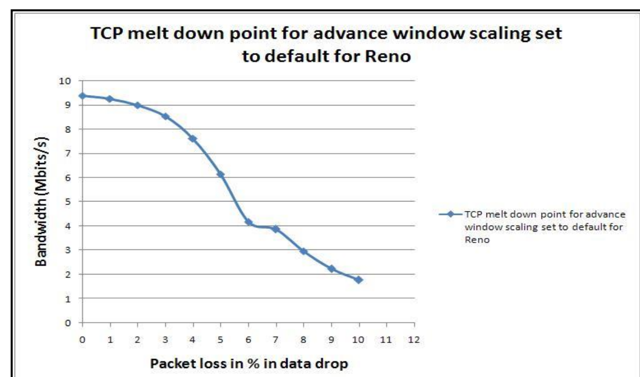


Figure 17: Performance of tcp_adv_win_scale when turned on

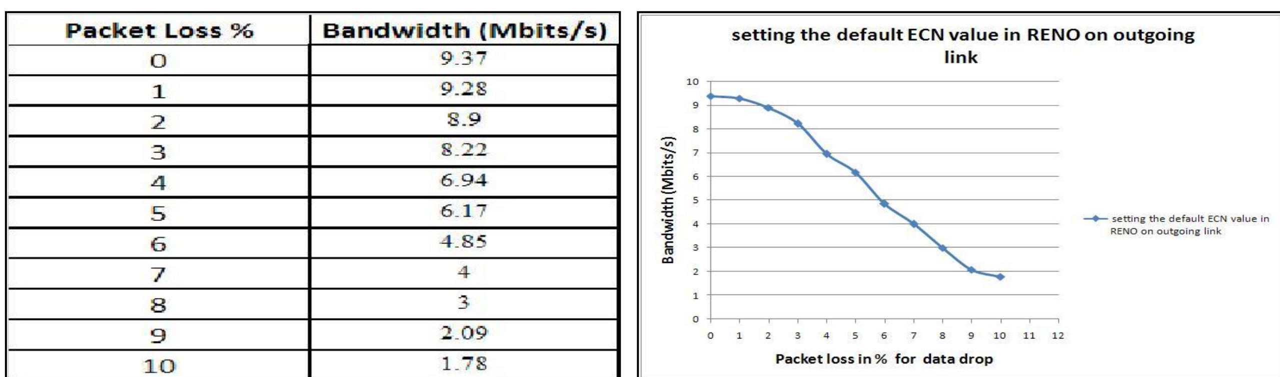
From the above figure, it can be seen that a drastic drop was noticed from 4% of data loss and ended up with only 1.77 Mbits/s for 10% data drop. As in our previous test, it was not able to

give a good performance for 10% data drop. However, when compared to figure 14, it performed slightly better from the beginning when packet loss was rather small.

5.1.6 Tests with different TCP variables: tcp_ecn

In the earlier test shown in figure 14 tcp_ecn was turned off (tcp_ecn=0). Tcp_ecn (Explicit Congestion Notification) is by default turned on (tcp_ecn=2). In this test, we wanted to observe the impact on performance when tcp_ecn was turned on and compare it with the performance shown in figure 14. This variable was turned on only in this particular experiment. In other cases it was set to 0.

tcp_ecn = 2 (default)



Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.28
2	8.9
3	8.22
4	6.94
5	6.17
6	4.85
7	4
8	3
9	2.09
10	1.78

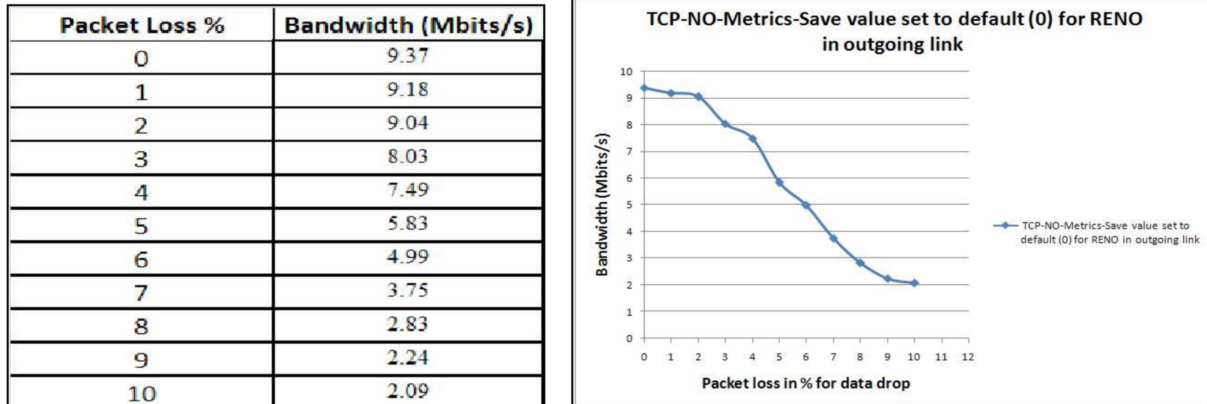
Figure 18: Performance of tcp_ecn value when turned on

Figure 18 shows the experimental result when tcp_ecn is set to 2. After analyzing the above graph it can be seen that turning on this variable doesn´t make it differs very much from the experimental result of figure 14. As before, performance drops drastically when data drop exceeds 4%.

5.1.7 Tests with different TCP variables: tcp_no_metrics_save value

The value of tcp_no_metrics_save was set to 1(save functionality disabled) in figure 14. But tcp_no_metrics_save value is by default set to 0(save functionality enabled). It controls a function that has got a function of remembering the last slow start threshold (ssthresh) when it is turned on. So we chose this variable to observe if it affects performance. A performance increase was expected from this experiment because it normally offers an overall performance improvement by remembering the previous results of the link, so it does not have to start slowly while searching for the congestion point. This functionality was turned off in all other experiments except in this experiment.

tcp_no_metriic_save = 0



Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.18
2	9.04
3	8.03
4	7.49
5	5.83
6	4.99
7	3.75
8	2.83
9	2.24
10	2.09

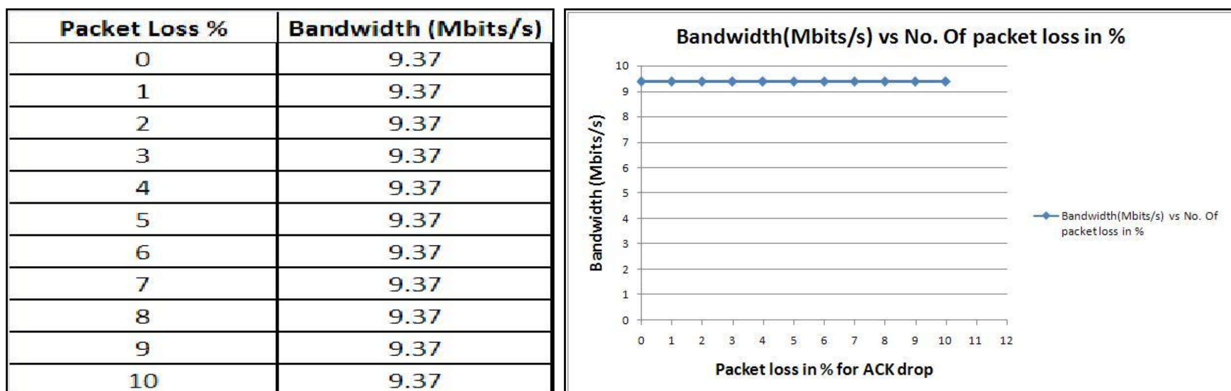
Figure 19: Performance of tcp_no_metric_save value when turned on

From the above figure (Figure 19) it can be observed that the performance was slightly better compared to Figure 14. For 10% of data drop the performance becomes 2.09 Mbits/s while in figure 14 for 10% data drop the performance was 1.93 Mbits/s.

5.2 Experiments with different congestion avoidance algorithms for ACK loss

5.2.1 CUBIC/RENO/H_TCP

All of three congestion avoidance algorithms have given the same experimental results when Acknowledgements are dropped. All TCP variables are turned off here for this experiment.



Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.37
2	9.37
3	9.37
4	9.37
5	9.37
6	9.37
7	9.37
8	9.37
9	9.37
10	9.37

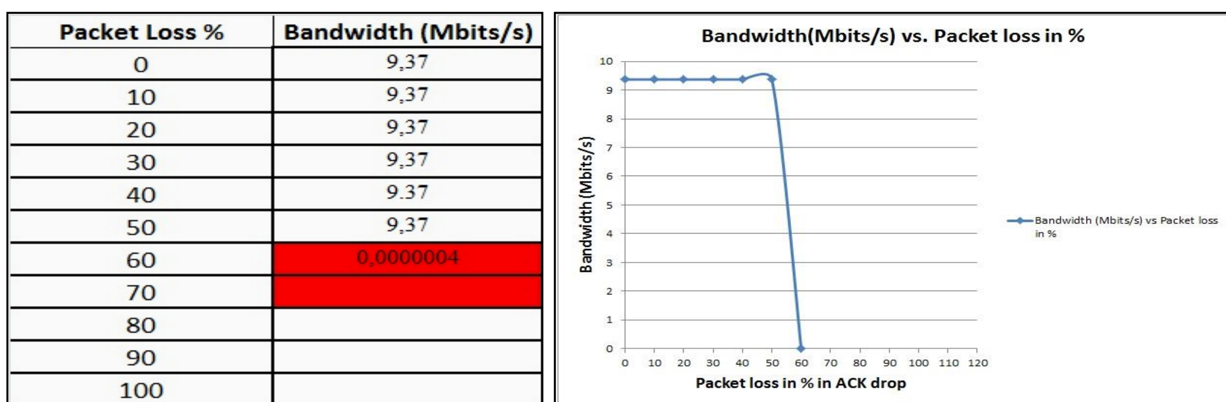
Figure 20: Performance for ACK loss with all three congestion avoidance algorithm (Cubic, Reno and H-TCP)

Figure 20 shows the performance while links are experiencing ACK loss. It can be seen that there was no change in performance regardless of whether the loss was 1% or 10% for all

three congestion avoidance algorithms. This experiment shows that even a relatively high amount of acknowledgement loss doesn´t have any impact on the performance. But to make this statement stronger further experiments with ACK loss have been done.

From experiment 5.2.1 it has been understood that small amount of ACK loss doesn´t have any impact on the performance. So the ACK loss was increased ten times from the previous test. That means 10%, 20%, 30%…100% of ACK loss was introduced to the network instead of 1%, 2%, 3%...10%.

Now, another test was done with congestion avoidance algorithm Reno; applying the increased amount of ACK loss. The obtained figure is given below.



Packet Loss %	Bandwidth (Mbits/s)
0	9,37
10	9,37
20	9,37
30	9,37
40	9.37
50	9,37
60	0,0000004
70	
80	
90	
100	

Figure 21: Performance for ACK loss with Reno when TCP variables were turned off

In figure 21, after increasing the loss, performance remains unchanged till 50% of ACK loss. At 60% of ACK loss the performance dropped to almost 0(i.e. 0.0000004).

Now, similar tests were done with the four TCP variables by turning them on separately in each test. To compare the performance of four TCP variables (when they were turned on) with figure 21, the following tests (5.2.2, 5.2.3, 5.2.4, 5.2.5) were done.

5.2.2 Tests with different TCP variables: tcp_window_scaling

The first test was done with tcp_window_scaling. It was turned on for this experiment.

tcp_window_scaling = 1 for ACK loss

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.15
50	9.37
60	0.00014
70	
80	
90	
100	

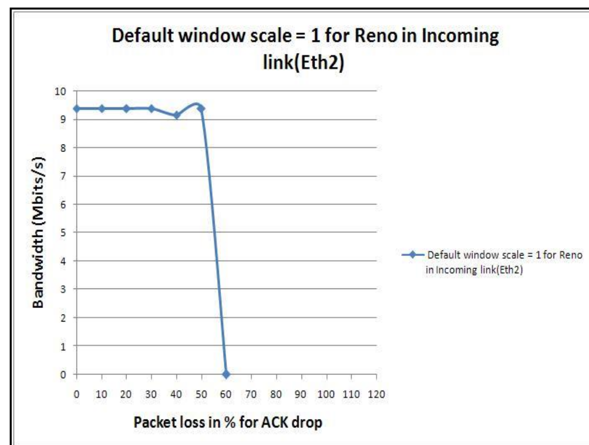


Figure 22: Performance of tcp_window_scaling when turned on

After increasing the loss, performance remained unchanged till 30% of ACK loss although performance decreased slightly when 40% of acknowledgments were lost. Similarly to figure 21, increasing the ACK loss to 60% caused performance to drop to almost 0(i.e. 0.00014).

5.2.3 Tests with different TCP variables: tcp_adv_win_scale

In this test tcp_adv_win_scale was turned on (set to 2). The meaning of the value has been discussed in 5.15 during the experiments of data loss. The same experiments were done for ACK loss to see its impact on the performance.

tcp_adv_win_scale = 2

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	8.18
50	9.37
60	0.0001535
70	
80	
90	
100	

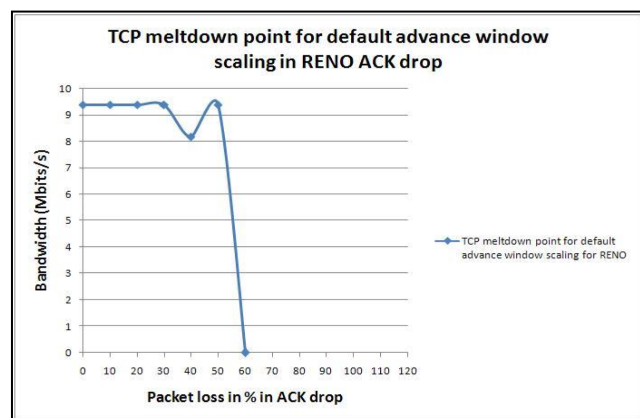


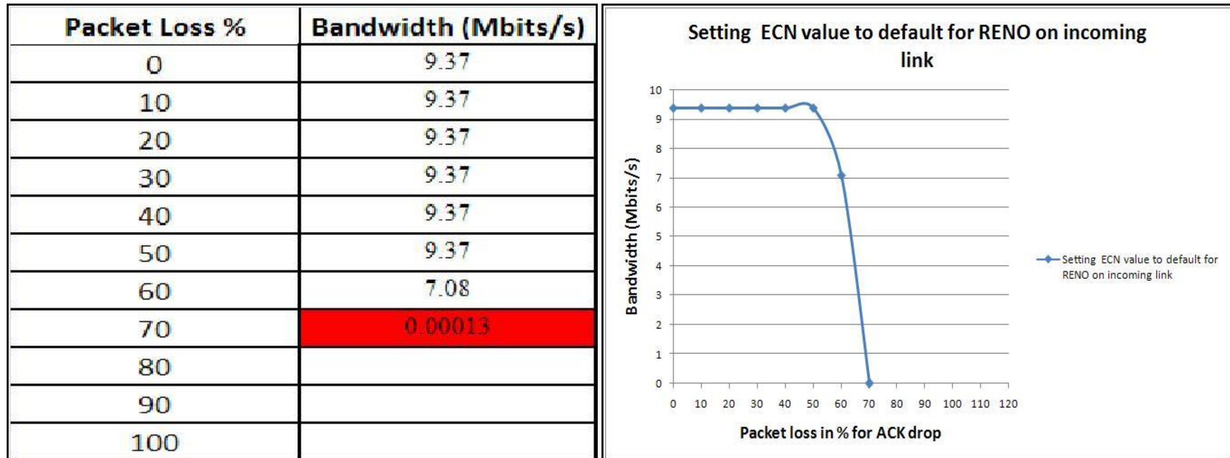
Figure 23: Performance of tcp_adv_win_scale when turned on

The above figure shows a drop in performance at 40% ACK loss, otherwise there is no significant difference from earlier experiments. As before, when we have reached 60% loss; performance was down to 0 (0.001535).

5.2.4 Tests with different TCP variables: tcp_ecn

In this test; tcp_ecn (Explicit Congestion Notification) was turned on, so that explicit notifications were delivered while congestion occurred.

tcp_ecn = 2 for ACK loss



Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.37
50	9.37
60	7.08
70	0.00013
80	
90	
100	

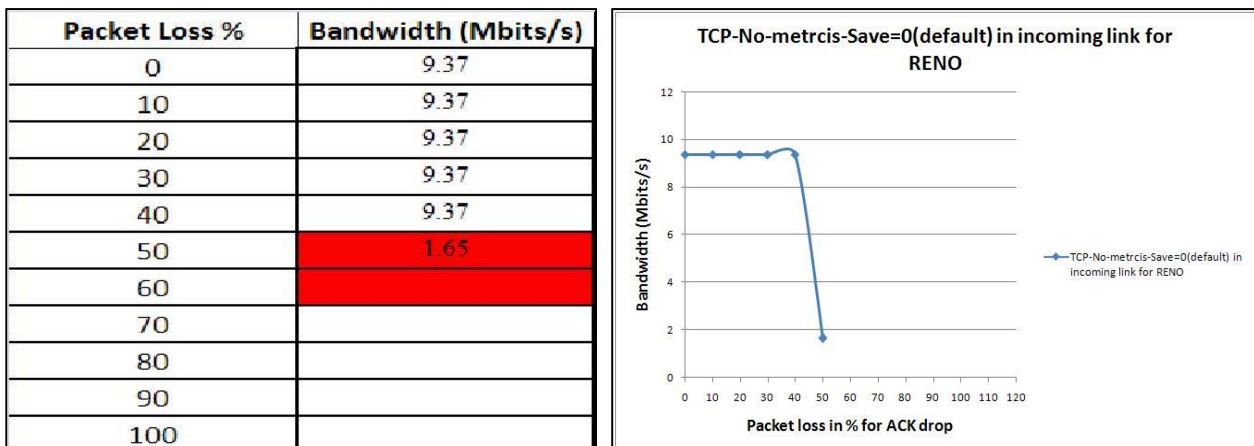
Figure 24: Performance of tcp_ecn when turned on

Performance was unchanged up to 50% of ACK loss i.e. 9.37 Mbits/s. For 60% of ACK loss performance dropped to 7.08 Mbits/s. However, compared with the figure 21 tcp_ecn was able to improve the performance at 60% quite significantly, but it wasn't able to handle 70% of ACK loss.

5.2.5 Tests with different TCP variables: tcp_no_metric_save value

Now, tcp_no_metric_save value was turned on to compare its performance with figure 21. The graph for ACK loss is shown in figure below.

tcp_no_metric_save = 0 for ACK loss



Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.37
50	1.65
60	
70	
80	
90	
100	

Figure 25: Performance of tcp no_metric_save value when turned on

In figure 25 it can be seen that performance drops already at 50% of ACK loss and goes down to 1.65 Mbits/s.

To conclude the experiments with ACK loss and as seen in the above graphs (Figure 21 to Figure 25) it is clear that changing the values of TCP variables doesn´t have much impact on performance when experiencing ACK loss.

Some extra experiments have been done with TCP variables both for data loss and ACK loss with Cubic in order to verify the results of Reno. But there were no major differences found in the results. These figures are presented in the appendix to avoid redundancy.

Chapter 6: Discussion

Chapter 5 contains description of all the experiments that were done during this thesis work. The results from these tests are presented in this chapter.

6.1 Outcome from the experiments when link experiencing Data loss

During data drop, the analysis of the three different congestion avoidance algorithms didn't show that much dissimilarity in performance drop from each other. A huge drop was observed for all the congestion avoidance algorithms. Only H-TCP showed slightly better performance (figure 26).

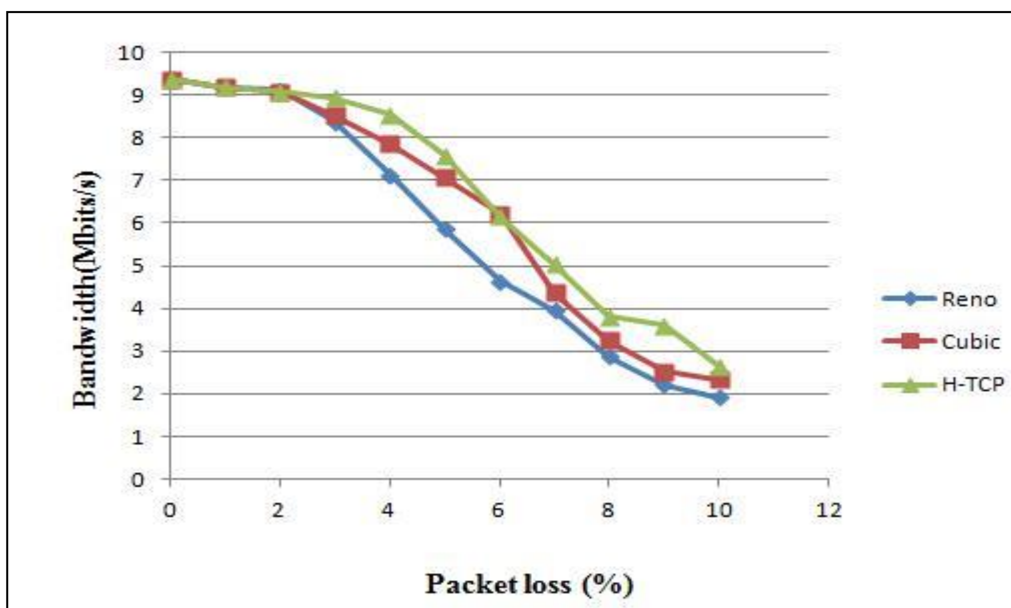


Figure 26: Performance of three different congestion avoidance algorithms while experiencing data loss

In figure 26 for 1% packet drop the bandwidth rates were 9.18, 9.19, and 9.19 for Reno, Cubic and H-TCP respectively. For 5% data drop these bandwidth decreased to 5.86, 7.07 and 7.58. While introducing 10% data loss the bandwidth rates declined to 1.93, 2.34 and 2.64.

Four different TCP variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save) were analyzed first with their default values and then by changing them one by one. Performance drop was quite similar for all four variables while they were turned off. A small raise in performance was observed when disabling tcp_no_metrics_save.

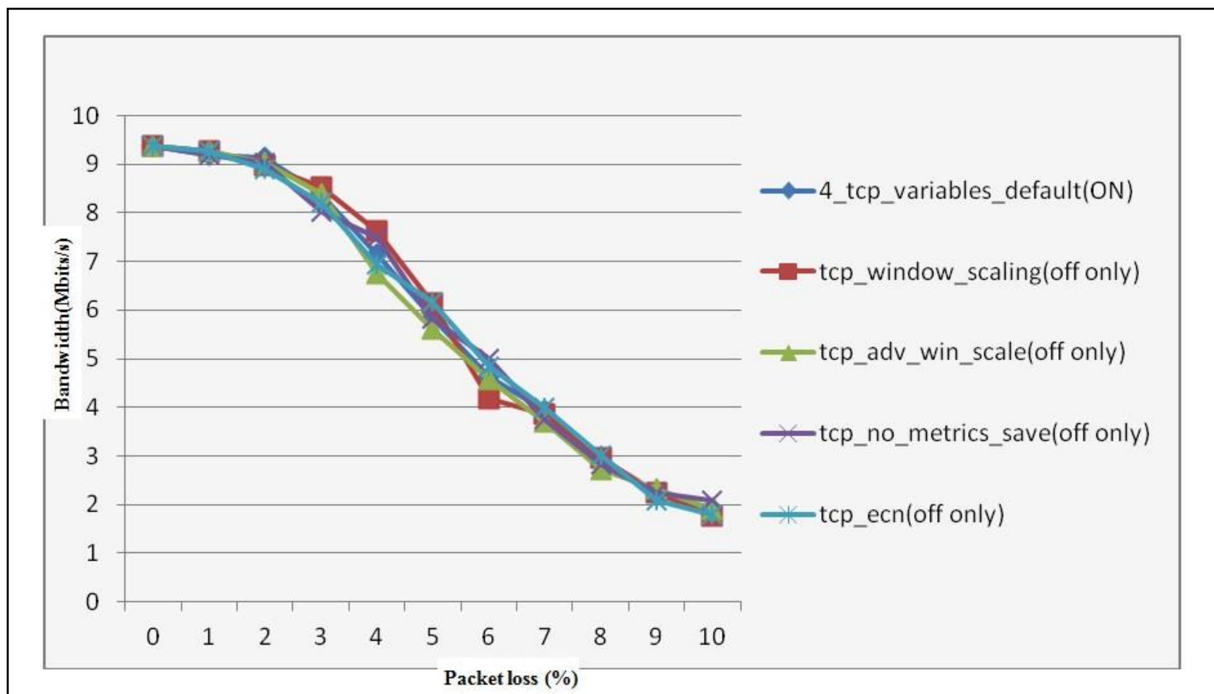


Figure 27: Comparison between four different TCP variables when turned off and when set to their default values while experiencing data loss

In figure 27 for 1% packet drop the bandwidth rates were 9.18, 9.26, 9.28, 9.18 and 9.28 with the different TCP variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save) set to default and turning them off (individually) respectively. For 5% performance drop the bandwidth rates fall down to 5.86, 6.13, 5.62, 5.83 and 6.17. While introducing 10% data loss the bandwidth rates declined to 1.93, 1.77, 1.91, 2.09 and 1.78 respectively.

6.2 Outcome from the experiments when link experiencing ACK loss

During the experiments, when the link was experiencing ACK drop, it could be seen that performance was not affected by a small number of acknowledgement drops. In the case of data drops, even for 1% packet drop, a decrease in performance was noticeable. While the ACK drop occurred, performance was unchanged for the link and remained constant at 9.37 Mbits/s even when the link experienced 10% of ACK loss.

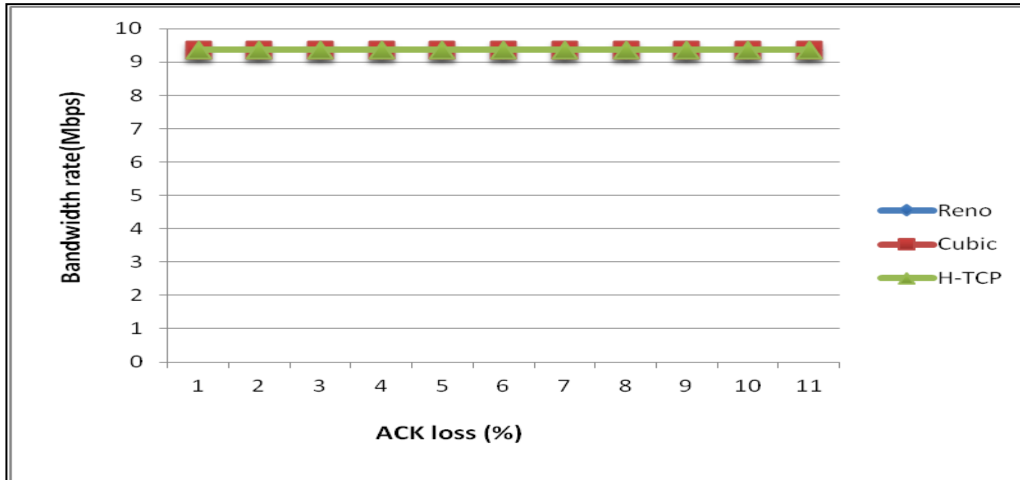


Figure 28: Comparison of the three different congestion avoidance algorithms in case of ACK loss

To see the performance when link experiences a high amount of ACK drops, we have increased the quantity of ACK drop to 10 times. Instead of 1%, 2%, 3%... 10% ACK drop rate, we observed the performance for 10%, 20%, 30%.....100% ACK drop. In these experiments; a temporary performance drop was noticed after 40% acknowledgement drop. At one stage congestion occurred in the network when the ACK drop was huge (70%).

The same experiments were repeated three times to ensure these results. Each time it showed performance drop after 40% acknowledgement drop and communication completely stalled at 60% acknowledgement drop.

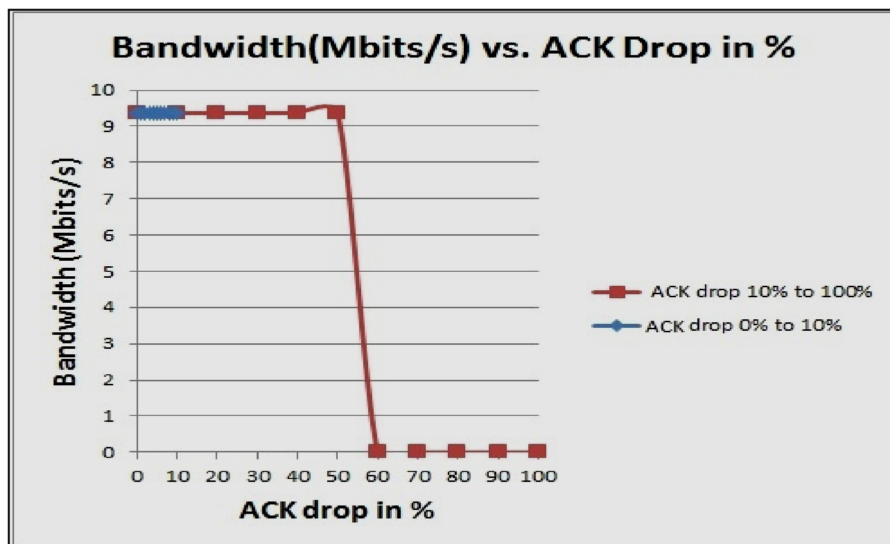


Figure 29: Performance of TCP Congestion Avoidance algorithms while experiencing different amount of ACK loss

We also tested to use different congestion avoidance algorithms (Reno, H-TCP, and Cubic) and when different TCP variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save) were set to their default values. This shows that performance doesn't fall with a small amount of ACK drops.

Experiments were also done by turning off TCP the variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save). This time a huge drop was observed when exceeding 50% of acknowledgement drop.

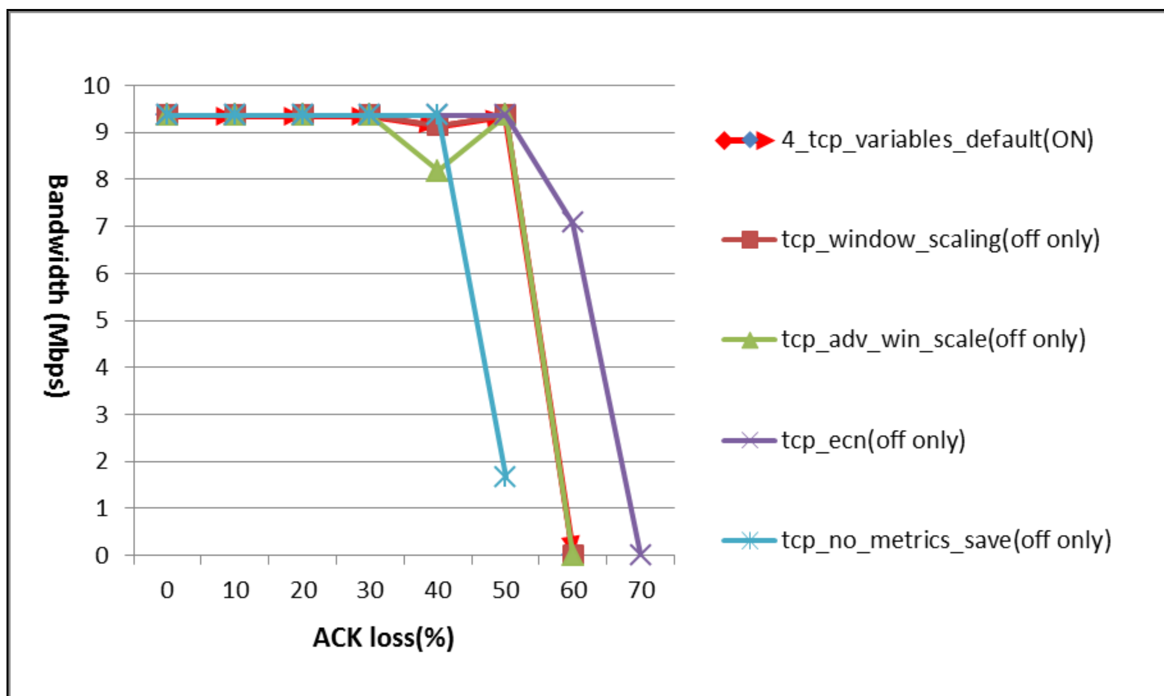


Figure 30: Comparison between four different TCP variables when turned off and TCP variables when set to their default values while experiencing ACK loss

In figure 30 when TCP variables were set to default (ON) bandwidth remain constant at 9.37 Mbps from 0% to 40 % ACK drop. For 50% ACK drop bandwidth declined to 0.0000004 Mbps. While TCP variables were turned off one by one, tcp_window_scaling and tcp_adv_win_scale showed a huge decrease in bandwidth rates with 60% ACK drop, at that time the bandwidth rates were 0.00014 Mbps and 0.0001535 Mbps respectively. With 70% ACK drop for tcp_ecn; bandwidth declined to 0.00013 Mbps. TCP_no_metrics_save showed a very low bandwidth rate which was 1.65 Mbps for 50% ACK drop.

6.3 Possible reason behind the drastic drop when link experiencing Data loss in RENO

The department of Information Engineering of University of Pisa of Italy has published a paper on the analysis of the TCP's behavior where they faced no packet loss [14].

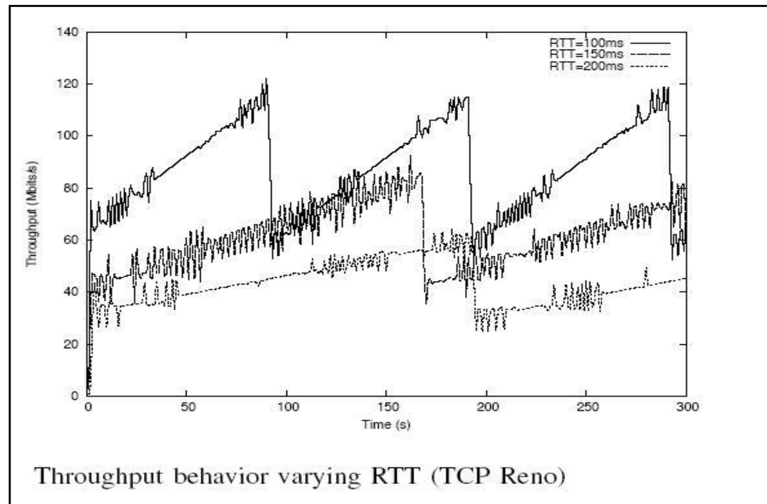


Figure 31: Performance during congestion in the general network

When Reno was running as the congestion control algorithm, even for a single packet loss, the throughput observed by them decreased from 120 to 60 Mbps which means that current transmission rate is temporarily decreased to 50% (Figure 31) and then increases linearly until another loss occurs.

The design of the congestion avoidance methods can be likely the reason behind the remarkable performance drop during heavy packet loss. Since transmission rate goes down to 50% for each lost packet and then the transmission rate is increased linearly. But after the next packet loss, transmission rate again decreases to 50% of the last value. There will be an imbalance in increase (linear) and decrease (cut in half) in transmission rate which will decrease the performance too fast. This can be one of the reasons behind declined performance of TCP.

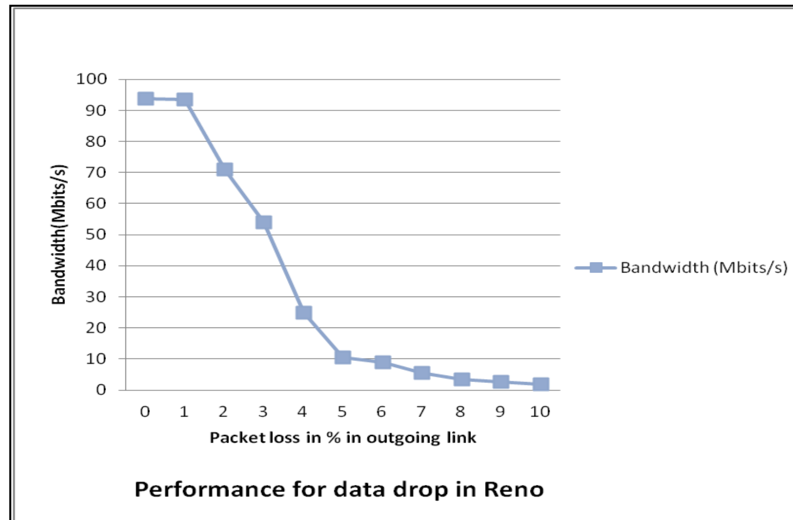


Figure 32: Drastic drop in performance in the experimental network

(In this thesis work)

Our findings are supported by [25] where the authors have simulated TCP Reno behavior during multiple packet loss. In Reno, upon receiving duplicate ACKs, the sender initiates a fast retransmission mechanism. When the fast retransmit mechanism signals congestion, the sender, instead of returning to Slow Start uses a Multiplicative Decrease Congestion Avoidance (MDCA) which is a part of the fast recovery mechanism. During transmission dropping two packets in the same window often leads to force the Reno sender to wait till retransmission timeout. During this drop, if the congestion window is less than 10 packets or the congetion window is within two packets of the receiver's advertised window Fast Recovery mechanism will be initiated. When three packets are dropped in the single window of data and the number of packets between the first and second dropped packets is less than 2+3W/4 (W is the congestion window just before the Fast Retransmit) the Reno sender will wait for a retransmit timeout. When four packets are dropped in a single window Reno sender have to wait for a retransmit timeout. With the increased number of dropped packets in the same window, the likelihood to wait till retransmits occurs increases, which disrupts data transmission and halts the transmission at a certain stage. When the fast retransmission mechanism is disrupted, the fast recovery mechanism will not be able to perform properly [25]. In our thesis work, by introducing packet loss we have dropped the packets randomly. So, there is a high possibility that more than one packet has been dropped in the same window. This indicates an inefficient use of the fast retransmission and fast recovery mechanism which can be a probable reason behind huge performance drop.

Chapter 7: Conclusion

In order to understand the reason behind TCP's drastic drop behavior with increasing packet loss, three different congestion avoidance algorithms (Reno, H-TCP, and Cubic) along with four different types of TCP variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save) have been analyzed. The department of Information Engineering of University of Pisa of Italy has mentioned in a paper that they have also observed a decrease in transmission rate with single packet loss which leads to significant performance drop; similar to the drastic performance drop we have observed in our thesis work.

In a small network we have analyzed the behaviour of TCP Reno by introducing packet loss. TCP Reno performed well when no packet loss was introduced, but by introducing 1% packet loss, the performance went down from 9.37 Mbps to 9.18 Mbps which means a 2% performance decrement and it linearly decreases. When reaching 10% packet loss, performance went down to 1.93 Mbps.

A similar type of experiment was done while introducing external ACK loss. A small amount of ACK loss did not affect performance at all; it remained constant at 9.37 Mbps. But a large amount of ACK loss for example 40% ACK loss caused performance to drop from 9.13 Mbps to 8.18 Mbps and with 60% ACK drop, performance went down to 0.00014 Mbps. Comparing Cubic and Reno, Cubic performed better. In Cubic the performance was constant at 93.7 Mbps till 60% of ACK drop. In Reno, performance decreased to 17.9 Mbps while link was experiencing 30% ACK drop and for 60% ACK loss, performance declined to 0% like Cubic. ACK loss is not as serious as loss of data packets since they are accumulative, meaning that a missing ACK will be covered for when the next ACK is delivered. However, when excessive ACK loss occurs, the sender has to retransmit the packets again, and it is likely that high ACK loss rate will rapidly increase the amount of retransmitted packets that, again will be lost and retransmitted, will create huge load in the network which likely is the reason for this performance drop.

We have compared three congestion avoidance algorithms with respect to packet loss; Reno's performance was the worst among the three algorithms. In Theory H-TCP was expected to be the best performing algorithm because this algorithm uses network resources efficiently, it is also able to acquire and release bandwidth fast in response to changing network conditions.

H-TCP is also suitable for using with simple and complex networks. Our analysis also showed that performance improved slightly with H-TCP.

We have also investigated how Four TCP variables (tcp_window_scaling, tcp_ecn, tcp_adv_win_scale, tcp_no_metrics_save) affected performance and they only slightly changed performance. Most of the experiments were done with Reno. Because Reno was the worst performing algorithm; if Reno can survive through the drastic drop in the performance with the increment of packet loss than other two algorithms will likely show better results in the same environment, in our experiments when experiencing packet loss and ACK loss. In the case of packet loss all the four TCP variables show almost the same drastic drop as before. Experiments were done by turning off the TCP variables for both packet drop and ACK drop; slightly better performance was observed (2.09 Mbps) with tcp_no_metric_save. On the other hand, a huge amount of ACK loss for example, 40% ACK loss caused performance to drop to 8.18 Mbps and 60% ACK drop to 0.00014 Mbps.

In conclusion we can conclude from all the experimental results that congestion control algorithm H-TCP performs slightly better than Cubic and Reno while the link was experiencing packet loss. A small amount of ACK loss doesn´t affect the performance while a large amount of ACK loss could not be handled by TCP.

Chapter 8: Future Work

The following is a list of recommendations for continued work on this topic.
In the network setup of our thesis, only the Linux (Ubuntu and Debian) operating systems were used. Instead of Linux, other operating systems like Windows and Mac can use to see their impact on bandwidth.

By using more powerful processors and hardware configurations, data processing speed will be increased which may reduce the amount of packet loss. It is also expected that it will have a positive impact on performance.

Four TCP variables were used in this thesis work. There are lots of other variables as well. The possible impact of other TCP variables may also have some influence of the bandwidth rate.

Other Congestion Avoidance Algorithms' behavior during multiple packet loss in the network will also be an interesting factor to discover in future. TCP may survive a single packet loss; but the real challenge is to cope up with the multiple packet loss.

References

1. Tunneling, http://www.tech-faq.com/tunneling.html [Acc. 19 Sep. 2011]
2. Honda, Osamu, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka and Junichi Murayama, *"Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency",* http://www.ispl.jp/~oosaki/papers/Honda05_ITCom.pdf [Acc. Date September 19, 2011]
3. Titz ,Olaf ,*"Why TCP Over TCP Is A Bad Idea",* http://sites.inka.de/sites/bigred/devel/tcp- tcp.html [Acc. 19 Oct. 2010]
4. Karlsson, Magnus Ullholm and MD. Ahasan habib, "*SSH over UDP*" http://publications.lib.chalmers.se/records/fulltext/123799.pdf [Accessed: November 4, 2010]
5. Canberk, Berk and Jaya Dhanesh,*"Tunneling TCP over TCP",* CHALMERS UNIVERSITY OF TECHNOLOGY, Göteborg 2004.
6. Lee, B.P., R.K. Balan, L. Jacob, W.K.G. Seah and A.L. Ananda, *"Avoiding congestion collapse on the Internet using TCP tunnels",* Computer Networks 39 (2002) 207-219
7. TCP congestion avoidance algorithm, http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm [Acc. 28 Oct. 2011]
8. Habibullah Jamal and Kiran Sultan*, "Performance Analysis of TCP Congestion Control Algorithms",* http://www.wseas.us/journals/cc/cc-27.pdf [Acc. 28 Oct. 2011]
9. Ipsysctl tutorial 1.0.4 Chapter 3. IPv4 variable reference http://www.frozentux.net/ipsysctl-tutorial/chunkyhtml/tcpvariables.html [Acc. 28 Oct. 2011]
10. About Wireshark, http://www.wireshark.org/about.html [Acc. 28 Oct. 2011]
11. Iperf, http://iperf.sourceforge.net/ [Acc. 28 Oct. 2011]
12. Netcat for Windows, April 10, 2009 19:52, http://joncraton.org/blog/46 [Acc. 28 Oct. 2011]
13. Robert Moris , "*TCP Behavior with Many Flows*", Harvard University, Atlanta, Georgia, October 1997.
14. Christian Callegari, Stefano Giordano, Michele Pagano, and Teresa Pepe, *"Behavior Analysis of TCP Linux Variants",* Dept. of Information Engineering, University of Pisa, ITALY

15. D.Leith and R.Shorten, "*H-TCP:TCP for high-speed and long-distance networks*", http://www.hamilton.ie/net/htcp3.pdf[Acc. 11 Dec. 2011]
16. tcp_adv_win_scale, tcp_ecn, tcp_no_metrics_save, tcp_window_scaling, http://linux.die.net/man/7/tcp [Acc.12 Dec. 2011]
17. Packet loss, http://swik.net/netem/Examples+of+Use [Acc.17 Dec. 2011]
18. MTU, http://www.dslreports.com/faq/7801[Acc.25 Dec. 2011]
19. Wireshark, http://sourceforge.net/projects/wireshark/ [Acc.4 Apr.2012]
20. Wireshark, http://technofriends.in/2009/01/22/how-to-using-wireshark-for-packet-analysis-part-1/ [Acc. 4Apr. 2012]
21. Netcat, http://netcat.sourceforge.net/[Acc. 13Apr. 2012]
22. Network Emulator, http://www.itrinegy.com/index.php/products/network-emulators[Acc. 15 Apr. 2012]
23. Ariane Keller, "*Manual tc Packet Filtering and netem*", ETH, Zurich, 20 July 2006.
24. D.E.Comer, *Internetworking with tcp/ip.* 5th ed., New Jersy: Prentice Hall, 2006, pp.212-213.
25. Kevin Fall and Sally Floyd, "*Simulation-based Comparisons of Tahoe, Reno, and SACK TCP",* ee.lbl.gov/papers/sacks.pdf [Acc. 5 Jun. 2013]

Appendix

1. Tests with different TCP variables in CUBIC for Data loss

1.1 tcp_window_scaling

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.28
2	9.17
3	8.7
4	8.29
5	6.8
6	5.48
7	4.34
8	3.69
9	3.15
10	2.13

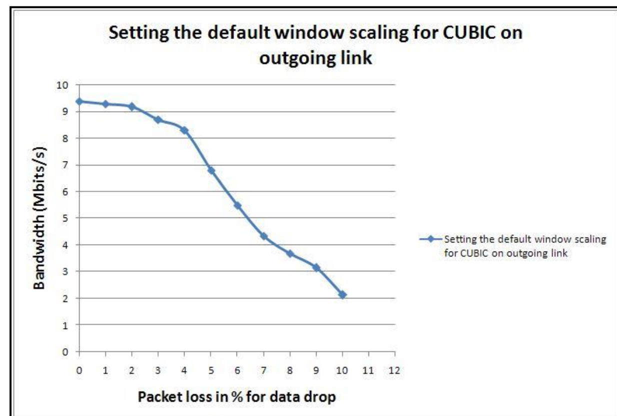


Figure 33: Performance of tcp_window_scaling when turned on

1.2 tcp_adv_win_scale

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.27
2	9.13
3	8.84
4	8.18
5	6.91
6	5.3
7	4.77
8	3.93
9	2.86
10	2.06

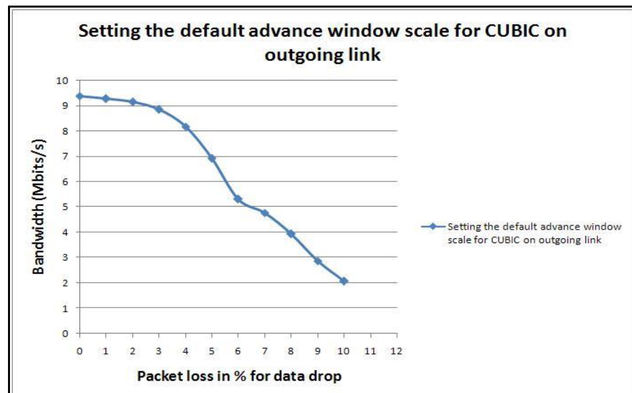


Figure 34: Performance for tcp_adv_win_scale when turned on

1.3 tcp_ecn

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.27
2	9.14
3	8.66
4	7.9
5	7.29
6	5.47
7	3.99
8	3.49
9	2.85
10	2.09

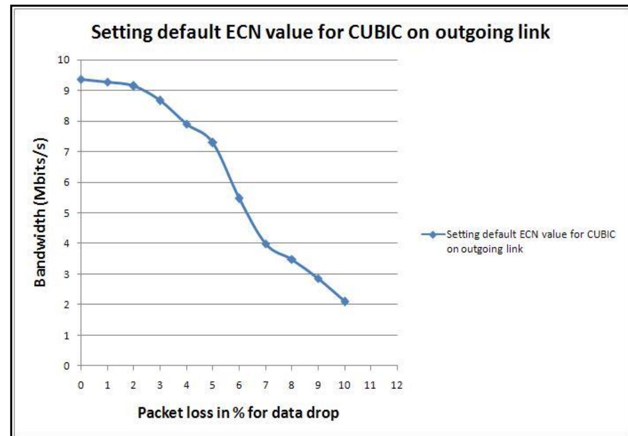


Figure 35: Performance of tcp_ecn when turned on

1.4 tcp_no_metric_save

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
1	9.28
2	9.15
3	8.92
4	7.73
5	7.11
6	5.7
7	4.03
8	3.07
9	2.84
10	2.11

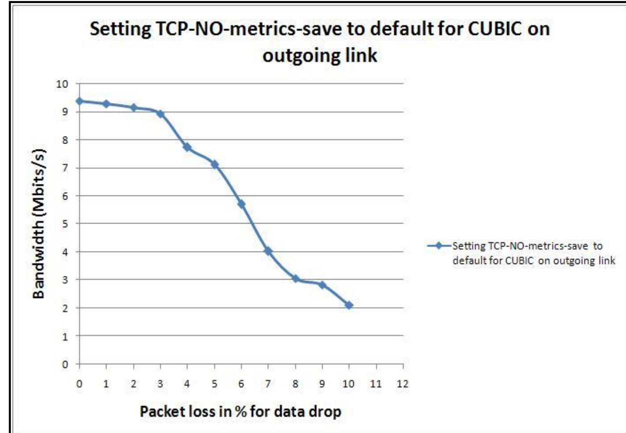


Figure 36: Performance of tcp_no_metric_save when turned on

2 Tests with different TCP variables in CUBIC for ACK loss

2.1 tcp_window_scaling

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.37
50	9.37
60	0.00013
70	
80	
90	
100	

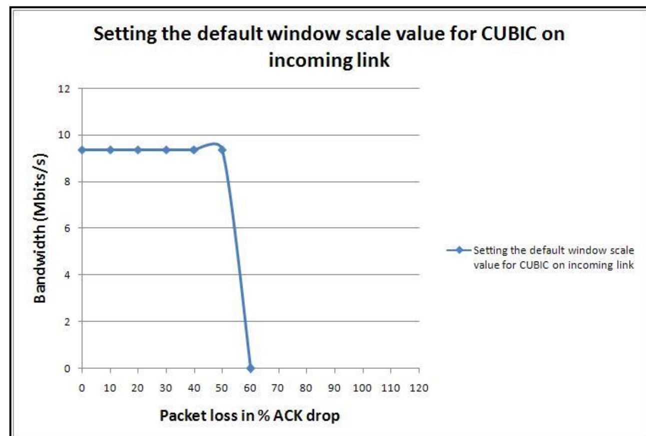


Figure 37: Performance of tcp_window_scaling when turned on

2.2 tcp_adv_win_scale

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.37
50	0.00107
60	
70	
80	
90	
100	

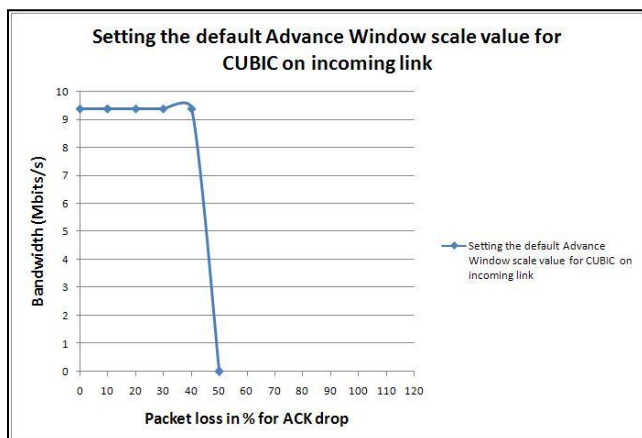


Figure 38: Performance of tcp_adv_win_scale when turned on

2.3 tcp_ecn

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.37
50	0.00039
60	
70	
80	
90	
100	

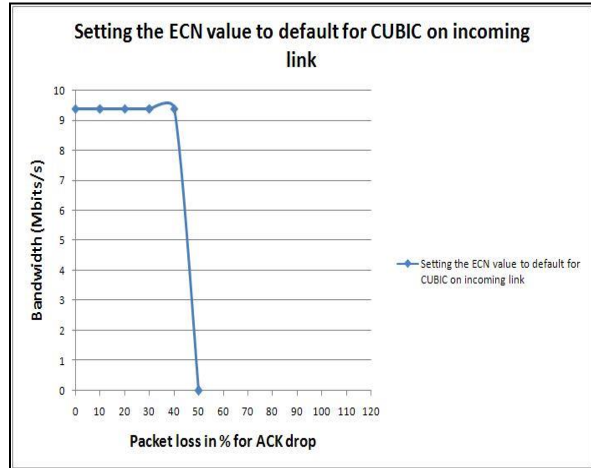


Figure 39: Performance of tcp_ecn when turned on

2.4 tcp_ no_metric_save

Packet Loss %	Bandwidth (Mbits/s)
0	9.37
10	9.37
20	9.37
30	9.37
40	9.23
50	8.04
60	0.0009241
70	
80	
90	
100	

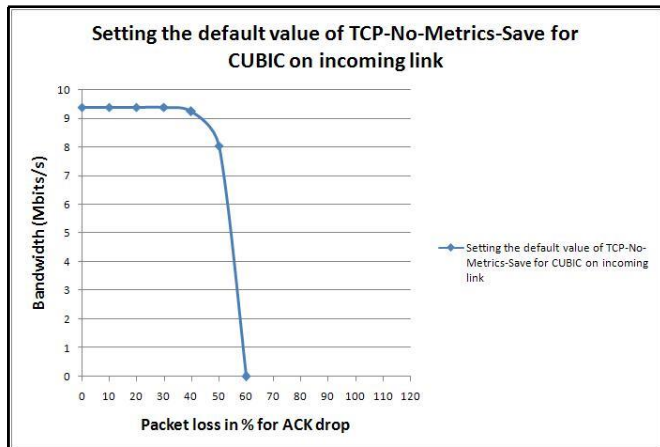


Figure 40: Performance of tcp_no_metric_save when turned on