UiO : Department of Informatics
University of Oslo

TCP Congestion Control Characteristics and Their Impacts in QoS over Mobile Broadband Networks

Mehraj Pakmehr

Master's Thesis Spring 2014

TCP Congestion Control Characteristics and Their Impacts in QoS over Mobile Broadband Networks

Mehraj Pakmehr

20th May 2014

Dedicated to the Memory of my dear and ever honoured Father.

Abstract

One of TCP's key task is to react and avoid network congestion episodes which normally arise in packet switched networks. A wide literature is existed in many different scenarios concerning the behaviour of congestion control algorithms and several congestion control algorithms have been proposed in order to improve performances. WLAN links have already been studied extensively in the literature. In this paper we focus on Mobile Broadband (MBB) networks that are in use today. We used NorNet Edge node which is connected to 3 different 3g ISP's (UMTS and CDMA2000). We also use three different TCP congestion control algorithms: TCP NewReno, TCP CUBIC as loss-based algorithms and TCP Vegas as delay based algorithms and try to see the impact of each TCP congestion control algorithm on the QoS characteristics. In the other word, we want to see how each MBB provider could affect the same TCP connection with same characteristics. We present QoS characteristics (e.g. Godput, delay) and discuss our observations. Our results could be used later for improvements in multi-path congestion controls.

Acknowledgement

I would like to express my appreciation to the following persons for their kind supports during this thesis:

- My sincere gratitude to Simone Ferlin-Oliveira, for accepting supervising me. During this thesis i learnt a lot from her and she kindly guided me through the right directions.
- My appreciation to Thomas Dreibholz my other supervisor, for all his efforts and supports. For his kindness helps in both technical and scientific way during this thesis.
- My deep gratitude to my internal supervisor, Paal Engelstad, as a knowledgeable professor. For his supports, thoughtful advices and encouragements during this thesis.
- A very deep gratitude to Amund Kvalbein, for his unlimited support and encouragements for me during this thesis. It was my real pleasure to have him by my side.
- Special thanks to Hårek Haugerud as a great and wonderful person and professor, During these two years of study he was the man who always supported me in all situations.
- Thanks to Kyrre Begnum as a knowledgeable professor, and great person. Every conversations that i had with him was a big lesson for me.
- A very special thanks to Ismail Hassan, whom he helped me so much during these two years. Words can not describe his kindness and knowledges.
- My biggest grattitude to my dear friend Forough Golkar, whom her helps was countless during my study periods. I want to thank her for all her supports and kindness that she had for me. she is the true friend whom i learnt so many things from her and she was always there for me.
- I would like to say thanks to the University of Oslo and Oslo University College for offering this Master degree program and all the facilities that they provided for me during my studies.

- Thanks to all my friends and fellow classmates for their company during this two years. I must say that nothing is comparable to finding new true friends.
- I would like to thank my lovely family, my mother whom she supported me everyday while there is so much distance between us. Also i want to thank my beloved brother and his lovely wife whom they always supported me through these whole two years.

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Problem Statement . . . 3

2 Background **5**
2.1 Transmission Control Protocol (TCP) . . . 5
2.1.1 Loss Recovery . . . 5
Fast Retransmit . . . 6
2.1.2 Flow Control . . . 6
2.1.3 Congestion Control . . . 6
Slow Start and Congestion Avoidance . . . 7
Fast Recovery . . . 7
TCP New Reno . . . 8
2.1.4 Enhanced Congestion Control Algorithms . . . 8
TCP Vegas . . . 9
TCP CUBIC . . . 11
2.1.5 Enhanced TCP Mechanisms . . . 12
Selective Acknowledgements (SACK) . . . 12
Forward Acknowledgements (FACK) . . . 13
Duplicate Selective Acknowledgements (DSACK) . . . 13
Forward RTO Recovery (F-RTO) . . . 16
The Eifel Algorithm . . . 16
2.1.6 Linux TCP . . . 17
Loss Recovery in Linux TCP . . . 18
2.2 Mobile Broadband Networks . . . 18
2.2.1 Universal Mobile Telecommunications System (UMTS) 19
2.2.2 CDMA2000 1xEV-DO (Evolution-Data Only) . . . 20
2.3 Related Work . . . 20

3 Approach **23**
3.1 Experiments design . . . 23
3.1.1 Requirements . . . 23
Quality of Services (QoS) variables . . . 23
Congestion control variants . . . 24
Different networks . . . 24
Traffic variation . . . 25
System settings . . . 26

3.1.2 Design . . . 26
Measurement tool . . . 26
Plotting tool . . . 27
3.2 Procedure and Collecting data . . . 30
3.2.1 Repetition . . . 30
3.2.2 Expected output . . . 30
3.2.3 Schedule of the experiments . . . 31
3.2.4 Collected data . . . 33
3.3 Analysing data . . . 34
3.3.1 Analysis tools . . . 34
3.3.2 Plots . . . 35

4 Results **37**
4.1 Implementation . . . 37
4.1.1 Testbed . . . 37
4.1.2 Measurements . . . 38
Measurement scripts . . . 44
Plotting scripts . . . 47
QoS calculation scripts . . . 52
Orchestration script . . . 56
Analysis plots . . . 57
4.2 Measurement results . . . 58
4.2.1 Bulk traffic . . . 58
Unlimited buffer size . . . 58
Limited buffer size . . . 59
4.2.2 Onoff traffic . . . 59
Unlimited buffer size . . . 60
Limited buffer size . . . 60
4.2.3 Stream traffic . . . 61
Stream with Unlimited buffer size . . . 62
Stream with Limited buffer size . . . 62

5 Analysis **75**
5.1 Bulk traffic . . . 75
5.1.1 QoS in different congestion controls in same network 76
Unlimited buffer evaluation . . . 76
Limited buffer evaluation . . . 80
5.1.2 QoS with same congestion control over different networks . . . 83
5.2 Onoff traffic . . . 85
5.2.1 QoS in different congestion controls in same network 85
Unlimited buffer evaluation . . . 85
Limited buffer evaluation . . . 89
5.2.2 QoS with same congestion control over different networks . . . 89
5.3 Stream traffic . . . 90

6 Discussion and future works **97**
6.1 Implementation overview 97
6.1.1 Metrics 97
6.1.2 Experiments 99
6.1.3 Measurement tools 99
6.1.4 Collected data 99
6.2 Results and Analysis overview 99
6.3 Future Works 100
6.3.1 Finding traffic manipulation by providers 100
6.3.2 Bufferbloats 100
6.3.3 Multipath congestion control 100
6.4 Potential weaknesses and improvement adjustments 101
6.4.1 Repetition of experiments 101
6.4.2 One-way delay measurement 101
One-way delay values from NetPerfMeter 102

7 Conclusion **103**

8 Appendix **105**
8.1 Appendix 1: On/Off traffic 105
8.2 Appendix 2: Bulk traffic 105
8.3 Appendix 3: Stream traffic 105
8.4 Appendix 4: Plot Scripts 105

List of Figures

2.1 Replication by the network in DSACK 14
2.2 False retransmission in DSACK 14
2.3 Early retransmit timeout in DSACK 15
2.4 ACK loss in DSACK 15
2.5 Example of 3G-UMTS RRC State Machine [26]. 20

3.1 Measurement tool design 28

4.1 NorNet Edge measurement node 38
4.2 Measurement setup overview 39
4.3 Location of the measurement node 40
4.4 Congestion window (CWND) plot 48
4.5 Sequence number plot 49
4.6 RTT plot 50
4.7 Throughput plot 51
4.8 Goodput plot 52
4.9 Bulk throughput sample 59
4.10 Bulk traffic with unlimited buffer size boxplots 64
4.11 Bulk traffic with limited buffer size boxplots 65
4.12 Onoff throughput sample 66
4.13 Onoff traffic with unlimited buffer size boxplots 67
4.14 Onoff traffic with limited buffer size boxplots 68
4.15 Stream 50% throughput sample 69
4.16 Stream 25% throughput sample 69
4.17 stream 50% traffic with unlimited buffer size boxplots 70
4.18 stream 25% traffic with unlimited buffer size boxplots 71
4.19 stream 50% traffic with limited buffer size boxplots 72
4.20 stream 25% traffic with limited buffer size boxplots 73

5.1 CWND sample of eth1, ppp0 and ppp1 in bulk traffic 77
5.2 RTT sample of eth1, ppp0 and ppp1 78
5.3 RTT and CWND of pp1 in Reno and Cubic with limited buffer 81
5.4 CWND sample of eth1, ppp0 and ppp1 in onoff traffic 87
5.5 RTT sample of eth1, ppp0 and ppp1 in onoff traffic 88
5.6 RTT-one way delay comparison of onoff traffic 90
5.7 CWND sample of eth1, ppp0 and ppp1 in stream 50 traffic . 92
5.8 RTT sample of eth1, ppp0 and ppp1 in stream 50 traffic . . . 93
5.9 CWND sample of eth1, ppp0 and ppp1 in stream 25 traffic . 94

5.10 RTT sample of eth1, ppp0 and ppp1 in stream 25 traffic . . . 95

List of Tables

2.1 Enhanced TCP Congestion Control Algorithms 9
2.2 Linux TCP parameters 17
2.3 Different TCP flavors emulation by Linux TCP 18

4.1 Send/Receive buffer size values 41
4.2 Bulk average Goodput per second with unlimited buffer . . 61
4.3 Bulk average Goodput per second with limited buffer 62

5.1 Bulk unlimited average values 78
5.2 Bulk limited average values 82
5.3 onoff unlimited average values 87
5.4 onoff Limited average values 89
5.5 stream 50% unlimited average values 91
5.6 stream 25% unlimited average values 92
5.7 stream 50% limited average values 93
5.8 stream 25% limited average values 94

Chapter 1

Introduction

The Internet has been in constant evolution since the early 1980s, where The Transmission Control Protocol (TCP, [3, 39, 61]) and the User Datagram Protocol (UDP, [62]) introduced. These two protocols carried more than 90% of the packets through the Internet between years 2001 and 2008, while TCP is dominant with more than 70% of this share [11]. TCP is a reliable, connection-oriented, full-duplex, byte-stream transport-layer protocol [54, 71] which supports congestion control [3] and flow control. Due to the reliability of the TCP, it has become the de facto standard used by many end-user applications, ranging from HTTP to bulk data transfer such as FTP.

TCP was designed for wired networks [61], where random Bit Error Rate (BER) is far more fewer than in wireless networks and the main cause of the packet loss is congestion, while in wireless networks the packet loss could be due to the limitations of radio coverage, user mobility, bad weather condition and etc. Hence, TCP misunderstands error loss as congestion loss which results in a way that sender backs off and the connection will face performance (Throughput) degradation [35]. It was also designed in a way that the connection establishment can takes place only in one path, meaning that establishing a TCP connection involves exactly one IP address per endpoint.

Due to the wide use of TCP in the Internet, it is thus crucial that TCP performs well over all kinds of wireless networks so the wired Internet be able to extend to wireless world.Wired networks and wireless networks are very different in terms of bandwidth, delay and reliability. There are several other factors that affect packet transmission in wireless networks such as:

- *Random Loss*
- *Larger Round Trip Time (RTTs)*
- *Bandwidth Limitation*
- *Handoffs*

- *asymmetric channel allocation* [7, 40]

Unimproved TCP interprets the packet loss in this environment as a sign of congested network and triggers congestion control system [37]. hence, the transmission rate slows down in order to reduce congestion which results in a considerably low throughput [48, 49].

As of today, Many improvements have been made on standard TCP (example: TCP Reno, TCP Tahoe [25] and TCP NewReno [27]) to improve its performance in wired networks. As a result, several TCP variants have been proposed which they have different approaches to how to deal with packet loss and what to do with congestion window, such as TCP Cubic [34], TCP Vegas [12], TCP Westwood[16] and TCP SACK [29]. However all these mechanisms and various versions do not work same as wired environment in wireless networks [14].

The research for finding a suitable TCP enhancement for heterogeneous networks (i.e. wired-wireless) is ongoing. In [9, 23, 51, 64, 68], the authors have compared different TCP enhancement schemes for mobile/wireless networks. They can be categorized as below:

- End-to-end connection
  The enhancement of TCP is applied at end hosts, and does not require any support from routers in between. Some notable proposals in this category are TCP Westwood [16], Freeze TCP [32], JTCP [72] and TCP Probing [69]

- Split connection
  TCP connection is divided to two connections. One is between the fixed host and base station/access point which uses the standard TCP connection and the other part which is between the base station/access point and mobile host which uses the proposed TCP schemes such as, Indirect TCP (I-TCP, [6]) and Mobile-TCP (M-TCP, [13])

- Localized link layer
  A link-level retransmission policy is performed at base station to recover from wireless losses quickly. Snoop [8] and its improved scheme [59] and Delayed Duplicate ACK (DDA) [70]

As a result of advancement in wireless technology and enhancements in TCP recent years have faced an ever-increasing usage of wireless networks ranging from wireless Local Area Networks (WLANs, IEEE 802.11) and Wireless wide-area networks (WWANs) such as 3G or 4G cellular systems. The number of handheld wireless terminals is increasing significantly. Today many people surf the internet by their mobile devices (e.g smartphones, tablets, laptops) via wireless technologies.

Based on the report by Ericsson[24], the number of mobile broadband subscriptions in the world passed 2 billion in 2013 and are expected to reach 8 billion by 2019. Also by the end of the same year, the report expects that

around 90 percent of the world's population will have access to Internet using 3G networks. These numbers together with the expected 10 times increase in mobile data traffic by the end of year 2019, shows why Mobile Broadband (MBB) has become much more interesting way for communication the recent years.

1.1 Motivation

The authors in [74] have discussed the transport protocol performance in IEEE 802.11 WLAN in details, therefore the focus of this thesis will be on Mobile Broadband networks.

In this thesis, we're trying to measure the QoS characteristics in Mobile Broadband networks and observe the TCP protocol behavior with different flavours - like, Congestion Controls, buffer sizes and etc. - with different MBB operators (different paths) to realize how is the imapct of the differences in TCP, in different network in multipath. In the other word, we are going to observe how different MBB networks behave with a same TCP connection and try to find out how does that difference impact the performance in multipath connection.

1.2 Problem Statement

This study focuses on measuring the Qos characteristics of different Mobile Broadband networks while they deal with same TCP connection (i.e. same Congestion control and same buffer size and etc.). Having the statement above, The following is the main Problem statement in this thesis:

How to choose the best type of TCP variant for different types of TCP traffic over different Mobile Broadband networks with different characteristics?

During this report, it is strived to address the following questions in order to reach our main problem statement:

- *How do differnet types of TCP Congestion Control (CC) schemes perform over different MBB networks?*
- *How do differnet types of TCP traffic (e.g. Bulk, App-limited) perform over different MBB networks?*

Chapter 2

Background

2.1 Transmission Control Protocol (TCP)

TCP [61] is a connection oriented transport protocol which guarantees a reliable end-to-end

delivery of data packets to the application layer. The application data that is submitted to TCP is divided to segments before transmission. TCP uses Automatic Repeat reQuest (ARQ) mechanism based on acknowledgments in order to achieve reliability. Each byte is numbered in TCP , Therefore the number of the first byte in a segment is used as a sequence number in the TCP header. In addition, A receiver, sends a *cumulative acknowledgement* (ACK) in response to the incoming segment, meaning that number of segments can be acknowledged at the same time.

2.1.1 Loss Recovery

When a segment is transmitted, TCP initiates a retransmission timer. The segment will be retransmitted , if the timer expires before the segment is acknowledged. This timer is called Retransmission Timeout (RTO) and its value is calculated dynamically based on the measurement of the time it takes from the transmission of a segment until the acknowledgement is received, referred as Round Trip Time (RTT). When a timeout occurs, the sender retransmits the lost segment and doubles the RTO (*exponential backoff*) [60].

By this method, A TCP sender can sample the RTT upon receiving each ACK. But if the sender receives an ACK which contains a sequence number of a segment which has alredy retransmitted, the sender can not distinguish whether the ACK is the receiver's answer to the original segment (e.g. reordered) or to the retransmitted segment.

There are two ways to overcome this problem. One is to use *Karn's Algorithm* [41], which ignores ACKs to retransmitted segments while estimating RTT. Another one is to use timestamp option field in TCP header. If both endpoints support this feature, then the current time will be included in timestamp option field in TCP header upon sending each segment. Therefore the correct and reliable RTT can be estimated.

i.e. sender can distinguish whether the ACK from receiver is for the retransmitted segment or for the original segment which has been sent for the first time [39].

Fast Retransmit

In order to avoid waiting a long time (i.e. one RTT) til the RTO expires for a missing segment, TCP employs a scheme called *fast retransmit*. It is possible that TCP receiver have some gaps of outstanding data in the received stream due to the fact that IP packets can get lost, duplicated or even reordered in the network. In this case, the receiver can not increase the ACK number, although it is receiving the new data. Therefore it sends an unchanged ACK number upon arrival of each new data. This ACK which its number is identical to previous ACK is called a *duplicate ack* (DupACK). This DupACK informs the sender that the receiver is missing one segment.

Fast retransmit assumes 3 consecutive DupACKs as the sign of packet loss. In this condition, the TCP sender resets the transmission timer and triggers fast retransmission.

2.1.2 Flow Control

In TCP, flow control ensures that a sender does not send more data than a receiver can handle. Since the data arrival at the receiver must be the exact same order that it was sent by the application, therefore, a TCP receiver should buffer out-of-ordered data until all gaps in the stream are filled before forwarding the complete and ordered stream to the receiving application. The receiver specifies a receive window (`rwnd`) in every ACK that it sends back to the sender to inform the sender that how many bytes it is allowed to send without overloading the receiver buffer [61].

2.1.3 Congestion Control

In 1986, the Internet faced its first congestion collapse. The end hosts transmitted as much data as they could, where the routers were not able to handle them, Due to the retransmission of the lost segments, the transmission rate did not get lower either. Van Jacobson proposed a a mechanism called congestion control [37] which has been implemented in TCP. Congestion control is TCP's attempt of not sending more data than what can be in flight on the path from sender to receiver.

The two values that limit the number of bytes a sender is allowed to transmit before receiving any ACK back are: the receive window (`rwnd`) and the congestion window (`cwnd`). The first is used for flow control and dictated by receiver, while the second is used to avoid network congestion and is an estimated value defined by the sender. The congestion window shows the estimated number of bytes that can be injected into the network without causing network congestion. A TCP sender must oblige both the `rwnd` and `cwnd` limits and calculates its send window (`swnd`) as:

$$swnd = min(rwnd, cwnd)$$

In TCP congestion control, the congestion window is gradually increased until packet loss is detected. By the detection of packet loss, the congestion window is quickly reduced in order to avoid causing congestion. this behavior is called *additive increase/multiplicative decrease* (AIMD).

Slow Start and Congestion Avoidance

There are two phases called *slow start* and *congestion avoidance* which they control the congestion window growth. In slow start phase, the capacity of the network is estimated by quickly increasing the number of sent packet per round trip time (RTT). This phase ends as soon as the cwnd reaches the threshold or packet loss is detected. which shows that the data rate is overloading the path capacity. In congestion avoidance phase, the cwnd is increased less quicker than slow start phase. The border between these two phases is *slow start threshold* `sshthresh` which can be referred as a value for sending data without congestion. When TCP establishes for the first time, it starts in the slow start phase, with the congestion window size of 1 segment and the slow start threshold set to a randomly high value. During slow start, the congestion window is increased by 1 Maximum Segment Size (MSS) for each incoming ACK. While during congestion avoidance, the congestion window is increased by 1 MSS per RTT.

The cwnd growth will stop when the sender detect that a segment is missing at the receiver, whether by receiving DupACK or if RTO expires and timeout occurs. In both cases, standard TCP interprets this loss of data as sign of network congestion and reduces both `cwnd` and `ssthresh`. As discussed in previous section, a TCP sender tries to recover from segment loss either after RTO expiration or after receiving 3 DupACK (fast retransmit). In both cases, the sshtresh will be reduce to half of the cwnd value.

$$ssthresh = cwnd/2$$

The difference is that, after timeout (i.e., RTO expiration), TCP will be forced back to slow start phase, in which the cwnd will be reset to 1. while after a fast retransmit, TCP goes to congestion avoidance phase by setting the cwnd equal to sshthresh.The standard TCP congestion control which is only based on RTO loss recovery and fast retransmit, normally referred as *TCP Tahoe* [25, 37].

Fast Recovery

The idea behind fast recovery is the fact that in fast retransmit, The receiving of DupACKs also means that the segment that the sender has sent, is now received and stored at the receiver, meaning that the network still is not congested. Therefore the sender knows that it can send new segments upon receiving every Dupacks and increments the cwnd by 1. This phase

ends by the reception of an ACK which acknowledges the new data (i.e., the ACK number has changed). At this point, the cwnd will be reduced to sshthresh which had been set before during the fast retransmit. The first TCP which uses both fast retransmit and fast recovery is TCP Reno[25, 38].

For the first and second DupACK, TCP uses the *limited transmit* which allows TCP to send a new data segment upon receiving the first two DupACKs, but the cwnd will not get incremented [2]. It proposed for scenarios such as very low-bandwidth paths in which the cwnd is too small, or when too many ACKs are lost. In this situations, A lost segment is not followed immediately by enough DupACKs which triggers the fast retransmit.

TCP New Reno

TCP Reno gets better path capacity utilization than TCP Tahoe, because of fast recovery mechanism. But it still has the limitation of retransmitting only a single segment per RTT, even in the cases that more than one segment were lost from the sent window. TCP Reno assumes that fast recovery phase will be finished with an ACK for the entire window. If for instance, two segments in a row get lost, then the fast recovery phase will be finished with an ACK that acknowledges only the first lost segments. Therefore the second lost segment never get retransmitted during fast recovery which results in retransmission timer to expire after the fast recovery phase.

TCP New Reno [27], have been proposed to cover this drawback of TCP Reno. It uses a mechanism which keeps track of the highest transmitted sequence number at the time that fast retransmit occurs. In this way, TCP New Reno stays in fast recovery phase until all the lost segments up to that highest transmitted sequence number are acknowledged.

2.1.4 Enhanced Congestion Control Algorithms

Many other congestion control algorithms have been proposed in order to achieve better utilization of the available path capacity. They differ from each other in terms of how to detect congestion. In addition to packet loss, which is a sign of a network congestion, other factors can also interpreted as a sign of this phenomenon. For instance an increase in RTT could be the result from congested router buffer. Therefore these different proposed congestion controls deal with different signs of congestion. Another difference is *how* they deal with congestion in terms of adjusting **cwnd** and **sshthresh** if it is detected.

The reason behind different proposed congestion controls algorithms is the fact that they are designed to improve the performance of TCP in different path environments. Some have been proposed for lossy wireless environments, while other work best in high speed networks or over satellite links with high latency. In Table 2.1, Kaspar [42] have summarized

all available congestion control algorithms in Linux (as of kernel 2.6.32) based on their functionality according to their estimated usage in Internet provided by [75]

TCP Version	**Congestion detection**		**Specialized for high**			**Internet**
Name	**loss based**	**RTT based**	**bandwidth**	**RTT**	**loss**	**usage [75]**
New Reno [27]	✓	-	-	-	-	17-25 %
BIC [73]	✓	-	✓	✓	-	14%
CUBIC [34]	✓	-	✓	✓	-	30%
H-TCP [50]	✓	-	✓	✓	-	0.49%
Hybla [15]	✓	-	-	✓	✓	-
Illinois [52]	✓	✓	✓	✓	-	0.76%
Low-Priority [45]	✓	✓	-	-	-	-
Scalable [43]	✓	-	✓	✓	-	1.86%
Vegas [12]	✓	✓	-	-	-	1.57%
Veno [31]	✓	✓	-	-	✓	1.22%
Westwood+ [20]	✓	-	-	-	✓	2.82%
YeAH [5]	✓	✓	✓	✓	✓	1.95%

Table 2.1: Enhanced TCP Congestion Control Algorithms [42]

TCP Vegas

TCP Vegas [12], is one of the proposed algorithms which relies on accurate RTT estimation. In TCP Vegas, an increasing RTT will be interpreted as sign of the congestion in the network. TCP Vegas uses the difference between the *expected* and *actual* flows rates to estimate the available bandwidth in the network. Hence, when the network is not congested, the actual flow rate will be close to the expected flow rate. By using this difference in flow rates, TCP vegas, estimates the available bandwidth and updates the window size accordingly. TCP Vegas, modifies three of TCP Reno techniques as below [47, 58]:

1. Congestion Avoidance:

   First, the sender calculates the expected flow rate as:

   $$Expected = \frac{CWND}{BaseRTT}$$

   Where :

   $$CWND = current\ window\ size$$

   $$Base\ RTT = minimum\ Round\ Trip\ Time$$

   Second, the sender calculates the actual flow rate as:

   $$Actual = \frac{CWND}{RTT}$$

Where:

$$RTT = The\ actual\ Round\ Trip\ Time\ of\ the\ packet$$

When the sender receives an ACK, it calculates the difference between the expected throughput and actual throughput as *Diff* using below equation:

$$Diff = (Expected - Actual) \times BaseRTT$$

TCP Vegas also defines two thresholds, α and β which their values is suggested to be (1, 3) in packet. The value of BaseRTTis updated if $Diff < 0$ since *Expected* value should exceed *Actual* value. According to *Diff*, the sender updates its window size based on Algorithm 1.

Algorithm 1 TCP Vegas Congestion Avoidance

if $Diff < \alpha$ **then**
  $CWND = CWND + 1$
else if $Diff > \beta$ **then**
  $CWND = CWND - 1$
else
  $CWND = CWND$
end if

2. Retransmission:

   If a single DupACK is received by the sender, it checks whether the difference between the current time and the sending time plus BaseRTT is greater than RTO or not, if it exceeds the RTO value, Vegas retransmits the lost packed without waiting for three DupACKs (Algorithm 2).

Algorithm 2 TCP Vegas Retransmission

if Current time $-$ (Sending time $+ BaseRTT$) $> RTO$ **then**
  send the lost packet and don't wait for 3 DupACKs
end if

3. Slow Start:

   During the *slow start* phase, TCP vegas uses the spaces between the ACKs to estimate the available bandwidth so it can set the *sshthresh* appropriately and respectively not exceed the available bandwidth.

In addition to TCP vegas, some other congestion control algorithms proposed later -like, TCP Veno [31] and TCP Illinois [52] which they mainly use the ideas behind TCP Vegas but not only depend on RTT estimation as a main sign of congestion.

TCP CUBIC

TCP CUBIC [34], is a proposed TCP variant with an optimized congestion control algorithm for high speed networks. It's an improved version of TCP BIC [73] with better window control and TCP friendliness. The main factor of TCP CUBIC is that its window size updates are independent of RTT and is only based on the time between two consecutive congestion events.

TCP CUBIC, is the default congestion control mechanism implemented in Linux kernel from version 2.6.26. It is similar to TCP Reno in some stages such as, Slow Start, Fast Retransmit and Fast Recovery. but differs when it comes to the adjustment of the congestion window. Instead of linear window growth function which TCP Reno uses, TCP CUBIC has a cubic window growth function as below:

$$W_{cubic}(W_0, t) = C(t - K)^3 + W_0 \tag{2.1}$$

Where:

$W_0 =$ Window size at the congestion event

$t =$ elapsed time since the last congestion

$K =$ the time required by the window to reaches the W_0 value with no loss

Which is:

$$K = \sqrt[3]{W_0 \times (\frac{\beta}{C})}$$

where: C is a constant (usually 0.4) and is called the **Cubic** parameter and β is another constant (usually 0.2) which is called multiplicative drop factor. In case of packet drop, the window will be reduced to $(1 - \beta)W_0$.

When congestion happens, the window size grows quickly with the sharp steep towards W_0 which is the window size at the time that congestion happened. When the window size is near to W_0 (steady state), it increases slowly. If the window size stays near W_0 for reasonable amount of time without any congestion, again the window size increases with the sharp steep to find a new steady state.

A high speed protocol can be referred as TCP-friendly if it is acting fair with standard TCP, i.e., takes equal amount of bandwidth when there is another standard TCP connection on the link. In order to reach link utilization and TCP friendliness, TCP CUBIC uses two window growth functions. The 2.2 function is the second function which is for when TCP CUBIC is operating in TCP friendly zone:

$$W_{TCP}(W_0, t) = W_0(1 - \beta) + 3\frac{\beta}{2 - \beta} \times \frac{t}{RTT} \tag{2.2}$$

When an ACK is received, both equations 2.1 and 2.2 will be evaluated in order to know the mode of the operation according to algorithm 3.

Algorithm 3 TCP CUBIC window growth function

if $W_{TCP}(W_0, t) > W_{cubic}(W_0, t)$ **then**
  Use equation 2.2
  Operate in TCP friendly region
else
  Use the cubic window growth function in equation 2.1
end if

Another high-speed optimized congestion control is TCP Illinois [52]. It uses packet loss as sign of congestion. However, it adapts the window relative to estimated queueing delay. Same as BIC and CUBIC, the outcome is the fast growth in window when there is no congestion and a slow growth, when congestion is probable.

2.1.5 Enhanced TCP Mechanisms

In addition to enhanced congestion controls which have been expressed in previous section, there are several other improvements to TCP which can be used in order to even more enhance the congestion control algorithms and TCP's loss recovery mechanism.

Selective Acknowledgements (SACK)

The problem with cumulative ACK (explained in 2.1), is that the receiver does not acknowledge the new segments that it has received after a segment loss and only sends dupACKs with the lost segment's sequence number to the sender upon receiving every new segment. therefore, the sender does not know whether the receiver has received new segments or not, meaning that it should wait either for an entire RTT to detect a lost segment, or retransmit the segments again which could be unnecessary. Selective Acknowledgement (SACK) [57], provides additional information to the sender about the segments that the receiver has received and those that it hasn't received by using additional information in the option field of ACKs. Hence, the sender has better overview of the receiver's stack and can make better decisions about which segments to retransmit, which dramatically reduces the number of unnecessary retransmissions [25].

Through the connection establishment, the SACK usage is negotiated via SYN segment between the sender and the receiver. If the negotiation is successful, then the receiver will be able to acknowledge a list of non-contiguous blocks of data by using the SACK blocks in the ACK's header option field. Each SACK block contains two sequence numbers, referring as *Left Edge* and *Right Edge*. Because of the limitation of 40 bytes in TCP options, only four SACK blocks can be included in each ACK. The first block of SACK always represents the latest segment than receiver has received.

In a SACK enabled connection, the sender keeps a list called *retransmission queue*, which is a list of all segments that have been sent but haven't been acknowledged yet. By receiving the ACK with SACK blocks from receiver, the sender marks the segments in retransmission queue as SACKed which have been specified in SACK blocks. Hence, unmarked segments previous to the highest SACKed segment are most likely lost and need to get retransmitted. A mechanism that uses SACK information to accurately estimate the outstanding segments in the network has been specified by RFC6675 [10] referred to a conservative SACK-based loss recovery algorithm for TCP.

Forward Acknowledgements (FACK)

The Forward Acknowledgement algorithm [56] looks for the most forward SACKed sequence number, referred as **snd.fack**, to get the better overview of the network and recover form situations where multiple segments are lost. Hence, the sender knows how many segments are missing and considers all unacknowledged segments (holes) between SACK blocks as lost and triggers fast retransmit if the number of holes is larger than the dupACK threshold.

FACK has the drawback in the situations where the holes are caused by packet reordering instead of packet loss. In such these situations, FACK behaves aggressively and triggers unnecessary retransmission for all holes. Operating Systems which use FACK e.g. Linux, disable FACK mechanism after packet reordering detection [42].

Duplicate Selective Acknowledgements (DSACK)

The Duplicate Selective Acknowledgements specified by RFC 2883 [28], allows a receiver to inform the sender about duplicate segments. The specification which SACK does not cover. A DSACK block is same as SACK block with the difference that the duplicate segment is mentioned as a single first SACKed segment and followed by the block that it is belongs to and additional blocks if there are any more.

Although DSACK itself does not provide any specific actions that the sender should implement, It lets the sender to distinguish between the events that the sender that only uses SACK is unable to do. The events that DSACK can express to sender can be summarized as below:

- Replication by the network:

  When a sender receives an ACK which includes a DSACK block stating to the segment that got never retransmitted, then the sender knows that the segment must got duplicated in the network. If the ACK is a dupACK, then the sender can make sure that the reason for dupACK is *replication* and not loss. Although the SACK itself can let the sender to identify the dupACKs that do not acknowledge

new data, but the DSACK option gives the sender a stronger basis for knowing that the dupACK does not acknowledge new data. Figure 2.1, shows the example in wich the segment has replicated in the network, and the receiver has notified the sender via DSACK option.

Transmitted Segment	Received Segment	ACK Sent (Including SACK Blocks)
500-999	500-999	1000
1000-1499	1000-1499	1500
	(replicated)	
	1000-1499	1500, SACK=1000-1500

Figure 2.1: Replication by the network in DSACK [28]., Sender sends segments with 500 bytes each.

- False retransmit:

  If a sender receives an ACK with a DSACK block referring to the segment that the sender has been retransmitted already, then the sender knows that the DSACKed segment was not lost and it just got reordered (i.e. the segment arrived more than 3 packets out of order) and caused an unnecessary retransmit. Figure 2.2, illustrate the scenario in which the segment got delayed and reordered, while the sender retransmitted it again. Hence, the receiver sends an ACK with DSACK option specifying that the retransmitted segment is a duplicate segment. In contrast, the SACK mechanism would not report the second received segment (duplicate one) as duplicate. hence, the sender wrongly thinks that the segment was lost.

Transmitted Segment	Received Segment	ACK Sent (Including SACK Blocks)
500-999	500-999	1000
1000-1499	(delayed)	
1500-1999	1500-1999	1000, SACK=1500-2000
2000-2499	2000-2499	1000, SACK=1500-2500
2500-2999	2500-2999	1000, SACK=1500-3000
1000-1499	1000-1499	3000
	1000-1499	3000, SACK=1000-1500

Figure 2.2: DSACK provides information to the sender, so the sender knows that the segment was reordered not lost [28].

- Early retransmit timeout:

  If the RTO of the sender is too small, then the retransmission timeout can occur, in which the sender retransmits the delayed segments knowing that the original segments were lost. However, the original segments of packets arrives at the receiver, resulting in sending

ACKs for received segment. After that, the retransmissions of the segments arrive which result in ACKs with DSACK which identify the duplicate segments. Hence, the sender learns that the RTO value is too short and it triggered false timeout. Figure 2.3 illustrates this scenario.

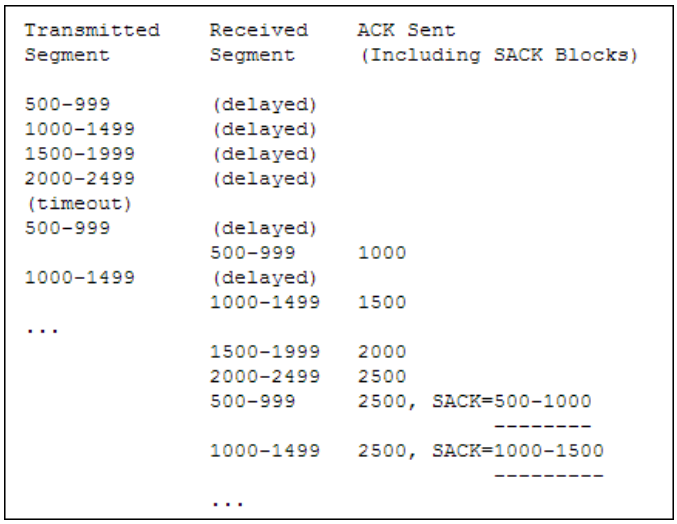


Figure 2.3: DSACK provides information to the sender, so the sender knows that the RTO is too small [28].

- Retransmit timeout because of ACK loss:

  If all the ACKs for an entire window is lost, the timeout will occurs. However, with the first ACK received after the timeout with a DSACK block, the sender will be notified about the duplicate segments received. SACK itself can not notify the sender about this matter. Hence, the sender won't be notified that none of the segments were dropped (Figure 2.4).

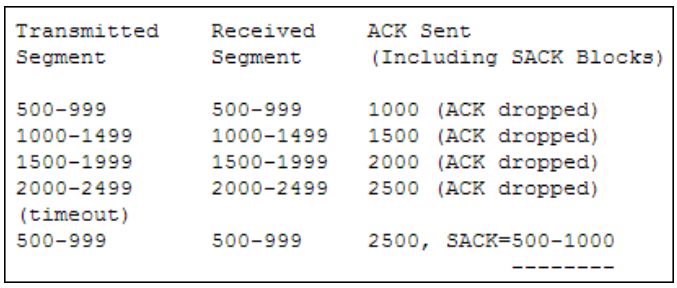


Figure 2.4: DSACK provides information to the sender about the duplicate segments received, so the sender knows that the segments were not lost [28].

One of the drawbacks of the DSACK is that it's unable to refrain the retransmission of a full window of segments when the false retransmit

timeout occurs. The reason is the sender is only able to know that the timeout is false *one RTT* after the false retransmission has occurred by receiving the first ACK containing DSACK block for the retransmitted segment from receiver (last line of Figure 2.4). However, during the elapse of that RTT the rest of the segments of the entire window are getting retransmitted continuously due to timeouts. The Forward RTO Recovery algorithm described in the following resolves this issue.

Forward RTO Recovery (F-RTO)

By the expiration of the RTO, TCP perceives that a loss has happened and not only enters to slow start stage, but also triggers the retransmission timeout, which results in unnecessarily sending the entire window of segments which are currently in-flight (outstanding). However, these timeouts could be because of normal delay spikes that even the most reliable links have and it even looks more common in wireless networks (e.g, loss of radio coverage or handover between base stations).

A TCP sender can use the F-RTO algorithm [66] to detect the false retransmission timeouts and refrain the whole window to get unnecessarily retransmitted. The main purpose of F-RTO is to send new segments rather than retransmitting the old segments in case of retransmission timeout occurs. Therefore, F-RTO looks for the next two incoming ACKs in order to perceive whether the timeout was false or not. If any of them were dupACK, then the F-RTO algorithms stops and TCP RTO recovery takes over. In contrast if the both incoming ACKs after timeout acknowledge new data, it means that the timeout was false and F-RTO continues sending new segments. Note that the first retransmission is unavoidable even with F-RTO. But it prevents further false retransmissions occurrence which unnecessarily occupy the link.

The Eifel Algorithm

The Eifel algorithm [55], is another enhancement to TCP for cases where false retransmissions and false timeouts occur frequently. It mainly uses timestamp option to know if the received ACK is the answer to original segment received or is the answer to the retransmission of the segment. Whenever the sender retransmit a segment for the first time, it stores its timestamp. If the sender receives an ACK with the related sequence number, it then checks its timestamp. If the timestamp value of the received ACK was smaller than the timestamp of the retransmitted segment, then the sender makes sure that the ACK is for the original segment and the retransmission was unnecessary. In addition, if there is only a single false retransmission occurs, the Eifel algorithm undo the congestion window adjustment by reverting it back to the value before entering loss recovery.

2.1.6 Linux TCP

Linux implements many of the recent TCP enhancements suggested by the IETF's RFCs. Some of these enhancements such as DSACK, FACK, TCP timestamp option, F-RTO recovery, and algorithms for packet reordering adaptation and undoing the wrongly reduced congestion window size still are not widely deployed in TCP implementations. Linux TCP also differs from the standard TCP New Reno and SACK TCP loss recovery schemes. Sarolahti and Kuznetsov, have summarized TCP features that differ from a typical TCP implementation in [65] as following:

- Retransmission timer calculation
- Undoing congestion window adjustments
- Delayed Acknowledgements
- Congestion window validation
- Explicit Congestion Notification

Linux provides a platform for testing the recent enhancements in an actual network. Therefore it is possible to to configure many of these mechanisms via `sysctl` command in Linux. Table 2.2, shows the parameters and their range along with their default values which previously described in subsections 2.1.4 and 2.1.5.

Parameter name	Value range	Linux TCP Default value
Congestion control algorithm	Table 2.1	CUBIC
Sack	0 , 1	1
FACK	0 , 1	1
DSACK	0 , 1	1
F-RTO	0 , 1 , 2	SACK-enhanced {2}
timestamps	boolean	1
$reordering_{min}$	[1 ... 127]	3

Table 2.2: Linux TCP parameters [42]

The result of possibly combining congestion control algorithms, loss recovery methods and performance enhancement mechanisms is a various TCP versions, also called *TCP flavors*. TCP Tahoe, TCP Reno and NewReno, SACK TCP and Eifel TCP are some examples of different TCP flavors. However, with Linux TCP , it is possible to emulate some of these TCP flavors via changing some of the described parameters in Table 2.2, since each TCP flavor has its own attributes (e.g, TCP New Reno, doesn't use SACK, FACK and etc..).

Table 2.3 shows the required values for specified parameters in Linux TCP, in order to reach the behavior of some of widely implementd TCP flavors such as, TCP NewReno, SACK TCP and Eifel TCP together with the Linux TCP values.

Parameter name	TCP NewReno	SACK TCP	Eifel TCP	Linux TCP Default value
Congestion control	NewReno	NewReno	NewReno	CUBIC
Sack	0	1	1	1
FACK	0	0	0	1
DSACK	0	0	0	1
F-RTO	0	0	0	SACK-enhanced {2}
timestamps	0	0	1	1
$reordering_{min}$	3	3	3	3

Table 2.3: Different TCP flavors emulation by Linux TCP [42]

Loss Recovery in Linux TCP

The Linux TCP sender, determines the number of currently outstanding segments in the network instead of comparing the congestion window to the difference of `snd.nxt` and `snd.una`. It then compares the number of outstanding segments to the congestion window in order to decide how much new data can be transmitted to the network and for recovering lost data segments. Note that Linux, tracks the number of outstanding segments in units of full-sized packets and does not compare cwnd to the number of transmitted octets. When SACK information is available, the Linux TCP sender uses the following equations to count the number of outstanding segments (`in_flight`):

$$\texttt{left\_out} = \texttt{sacked\_out} + \texttt{lost\_out} \tag{2.3}$$

$$\texttt{in\_flight} = \texttt{packets\_out} - \texttt{left\_out} + \texttt{retrans\_out} \tag{2.4}$$

In equation 2.3, `sacked_out` is the number of all SACKed segments and `lost_out` is the estimated number of lost segments in the network. In 2.4, `packets_out` is the number of all originally transmitted segments (i.e, `snd.nxt - snd.una`) and `retrans_out` is the number of retransmitted segments, as its name shows. Determining the `lost_out` value is not as easy as the other parameters - since they are computed based on sent data and receiving the SACK information from the returning ACK - and depends on selected recovery method based on algorithm 4.

2.2 Mobile Broadband Networks

From a few years ago the mobile communications technology changed the way people communicate. looking to the history of this evolutionary path, we faces to the first generation of mobile broadband (1G) which accomplished the basic mobile voice, backs into analog cellular technologies goes back to *1980s*. The second generation (2G) has introduced capacity and coverage with short messages and low speed data. The *CDMA2000 1xRTT* and *GSM* are kinds of 2G technologies while 2G technology became available in the *1990s*. The third generation (3G) have been initiated data at higher speed which *UMTS-HSPA* and *CDMA2000 EV-DO* were the primary 3G

Algorithm 4 Linux TCP loss recovery method

```
if SACK is enabled then
  if FACK is enabled then
    Count all un-ACKed data between SACKed blocks as lost
                                  # Aggressive FACK-based recovery
  else
    Consider all un-ACKed data as outstanding
                                  # Conservative SACK-based recovery
  end if
else
  Increase sacked_out by one for each incoming dupACK
                                  # TCP New Reno
end if
```

technologies. This evolution combined with portability of mobile broadband (MBB) networks led into debatably increase in impertinence of MBB as a communication platform. The usage of MBB varies in mobile devices such as smart phones and tablets with fast speed and high capacity MBB networks [44, 46] .

2.2.1 Universal Mobile Telecommunications System (UMTS)

The *universal mobile telecommunications systems* (UMTS) is the third generation mobile broadband network based on GSM and have 3GPP (3rd Generation Partnership Project) as the standardization.

Two main subsystems of 3G-UMTS are:

- **Terrestrial radio access networks (UTRAN)**
- **Core Network (CN)**

The UTRAN communication network known as the 3G while contains *radio network controller (RNC)*, *user equipments(UEs)* and *base stations* which is known as (NodeBS).

The RNC has responsibility of controlling one or more NodeBs which the RNS and NodeBS can be the same device in the network, Although there is no necessity for them to be physically separated , but the have different logical interface. The RNc make UEs to access to the CN and based on the UEs state transitions the NC keeps track of UEs radio resource control (RRC).

While the UEs are connected the RNC will assigned a RRC-state based on the its state on its NodeBS. Typically three RRC-state are available: **IDLE** or *DISCONNECTED*, forward access channel (**CELL-FACH**), and dedicated channel (**CELL-DCH**).

CELL-DCH is a dedicated channel with high bandwidth, while the **CELL-FACH** is a low-bandwidth channels. The RRC state are controlled by two timers and one threshold of data rate as illustrated in Figure 2.5.

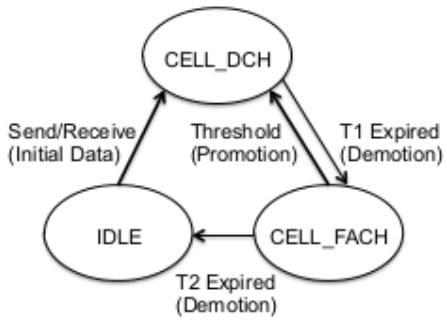


Figure 2.5: Example of 3G-UMTS RRC State Machine [26].

When a data threshold is exceeded and state demotion are controlled by timeout, the state promotions are perform T1 and T2 shows in Figure 2.5. At the time that data arrives, depend on the RRC configuration the the RRC-state transition can be **CELL-FACH** or **CELL-DCH**, while the **CELL-DCH** are limited in NodeBs and dedicated in single UE[26].

2.2.2 CDMA2000 1xEV-DO (Evolution-Data Only)

The *CDMA2000 1xEV-DO* is a 3G telecommunication standard which is an evolution of *CDMA2000* and has been standardized by 3rd Generation Partnership Project 2 (3GPP2), while *CDMA2000 1xEV-DO Rev.A* is a revision of EV-DO that adds several addition to the protocol with keeping the compatibility with revision 0.

The *CDMA2000* created with different components compare to the *UMTS* networks, nevertheless there are similarities in components number and functionality such as UEs RRC is like *UMTS*.

If the High Speed Packet Access (HSPA+) is available the *3G-UMTS* supports maximum data rates, up to 21Mbit/s in downlink and 11Mbit/s in uplink, while *3G-CDMA2000 1xEV-DO Rev.A* supports 3.1Mbit/s and 1.2Mbit/s in downlink and uplink respectively[26].

2.3 Related Work

From the 1990s the Quality of Service (QoS) characteristics of wireless networks have been a point of interest which so many research went around it.

The authors in [74] studied IEEE 802.11 WLAN and proposed a discussion about the transport protocol performance in wireless networks.

In [17], the authors focused on the TCP performance in 3G networks and considered the rate and delay differences.

The aouthers in [19], focused on applications with different QoS requirements in CDMA networks. They studied different TCP performance parameters, such as throughput, round-trip times and loss rates, and they introduce they estimate the bandwidth to achieve application Qos.

Tan et.al. performed a measurements in 3G networks to assess temporal and location of network characteristics and proposed analysis study of the TCP performance impact on the Application Layer [67].

In [53], a measurement study has been provided to observe the relationship between the MAC layer in Code , Division Multiple Access (CDMA) and TCP with different congestion control mechanisms.

Chapter 3

Approach

This chapter provides an overview of the methods and techniques used in this research. It covers the considered ways that tries to answer the problem statement following by design of experiments needed to reach the operationalization. Furthermore the methodology and the testbed used in this study will be covered in this chapter followed by the expected results and analysis methods in order to be bale to interpret the differences and the way to best answer the problem statement.

3.1 Experiments design

In order to properly answer the problem statement, a proper experimental design should be considered. A testbed should be considered which consists of two machines. one can be considered as *Client* and the other one as *Server*. According to the assumption that different Mobile Broadband (MBB) network paths should be observed, therefore there should be different connection paths provided by different MBB providers for either Client or Server (depending on the experiment scenario). A TCP connection could be established on different network paths between client and server and specific traffic type could be generated between two nodes depending on whether the direction of the traffic is from server to client or from client to server (i.e, *Downlink* or *Uplink*).

3.1.1 Requirements

The experiments should be designed in a way that they represent a real usage experience over MBB network (e.g, simulating the type of traffic and usage that a user with handheld MBB terminal can experience. Upon each connection, the different parameters as following could be used in order to construct this experiment.

Quality of Services (QoS) variables

It is crucial to define the variables that should be measured and compared in these sets of experiments. Thus the QoS variables and their explanations

could be as following:

- *Throughput:*

  The rate of successful message delivery over a network path. It usually measured in bits per second or Bytes per second.

- *Goodput*

  The application level Throughput could be referred as *Goodput*. In Goodput, the amount of data that the application is receiving excluding protocol overhead and retransmitted data packets is considered, rather than the total amount that the network interface receives successfully (i.e, Throughput). The goodput is always lower than the throughput.

- *RTT*

  The time that it takes from the transmission of a segment until the acknowledgement is received. Described in section 2.1.1 on page 5.

- *One-way Delay*

  The time that the packet spends in travelling across the network in one way (e,g uplink or downlink).

Congestion control variants

Since the problem statement focuses on impact of different TCP congestion control mechanisms in different networks, therefore it is needed to conduct the experiments in a way that measurements could be categorized in specified congestion control algorithms, meaning that experiments (i.e, TCP connections described above) could be run with the same type of congestion control algorithm each time on every network paths available for the experiments and the behaviour of each connection with same congestion control algorithm could be observed and compared.

The selection of the congestion control algorithms for experiments could be based on their congestion detection type provided in Table 2.1 on page 9. Hence, by having mixture of *loss-based* and *RTT (delay) based* congestion control algorithms as variants in our experiments, the comparison would cover both types of congestion control mechanisms and makes the analysis stronger by comparing the behavior of the loss based and delay based algorithms together in the same condition and path.

Different networks

As noted in the problem statement, we are interested to see the impacts of different congestion control algorithms in different MBB networks. Therefore we should set the experiments in a way that it runs the same TCP connection with same congestion control algorithm across different MBB networks. The selection of different MBB networks could be from

different MBB providers in Norway , preferably with different technologies as described in . By choosing different technologies and comparing them under same condition (i.e, congestion control and etc..), we can conclude whether the difference in technology is involved in the impact of congestion controls on our specified QoS parameters or not.

Traffic variation

In order to simulate the usage of a user and compare the QoS results of different congestion control mechanisms over different networks, various traffics must be generated and observed. These traffics could represent the user side usage working with mobile terminal equipped with Mobile Broadband technology. The proposed traffics could be as following:

- Large data transfer (*bulk traffic*)

  Emulates the downloading or uploading a large file by the user. In this type of traffic the delay is not an important factor since it is not application sensitive traffic. The rate of the transfer could be important factor to monitor. This kind of traffic could also be referred as long flow traffic in which, the sender and receiver try to reach to maximum available capacity of the link i,e. the sender sends as much segments as possible unless the receiver or the routers on the path can't handle the rate of transferring packets resulting in packet loss or RTO and triggering the congestion control mechanism of the sender and backing of from saturating the link.

- web surfing (*onoff traffic*)

  Represents the web surfing, in which concurrently downloads or uploads a certain amount of data for a specified time and then stays idle for some other specified time. Aiming at emulating the users web surfing, while downloading a web page in limited amount of time(e.g, 1 second) and then stay idle for the time more it took for downloading the page and reading it for instance. This type of traffic could also referred as short flow traffic. During the transmission time, the sender tries to send as much segment as possible (i.e, the link capacity has been reached and saturated).

- Streaming (*Application limited traffic*) Normally when applications such as video/audio streaming servers, video/audio chats, Voice over IP (VoIP) and etc. stream the data, the rate of the transferring segments is limited and has a fixed threshold which depends on the codec and bitrate of the stream, the segment transferring rate differs. An example of bandwidth and rate consumption of different codecs in VoIP traffic is provided by Cisco in [18]. Since the rate is limited by the application, therefore this kind of traffic could be referred as application limited traffic. In application limited traffic, the sender does not saturate the network by aiming to reach the maximum transferring rate.

System settings

As described in 2.1.6 in Background Chapter, It is possible to tune and configure Linux TCP parameters in order to properly combine and simulate the behavior of some of the well known and widely implemented TCP flavors. However, In this thesis we're mainly interested in congestion control algorithms and also the buffer size of the receiver. The latter will be considered mainly because of simulating the current wireless handheld terminals in which, the amount of memory that the TCP stack could use for its buffering is low due to the resource limitation in these devices comparing to other devices like - laptops, etc. This buffer size value could be set via the `sysctl` command in Linux.

3.1.2 Design

By assuming the requirements mentioned above, if we consider B as the buffer size, T as the traffic type, C as congestion control algorithm and N as the network, therefore the definition of each experiment defined as Exp could be as:

$$Exp_{(B,T,C,N)} \begin{cases} B \in \text{[limited, unlimited]} \\ T \in \text{[bulk, onoff, stream]} \\ C \in \text{\{loss based, delay based\}} \\ N \in \text{[Network A, Network B, Network C]} \end{cases} \tag{3.1}$$

For buffer size in equation above, *limited* could represent the wireless handheld terminal with limited memory and *unlimited* could be a value much higher than limited value (e,g. equal to the amount of data in bulk transfer in Mega Bytes).

Knowing the structure and variables of the experiments, A tool could be designed and implemented in order to provide the desired experiment. The design of the measurement tool is described as following.

Measurement tool

With the assumption of the required parameters described above, a tool must be implemented which runs the measurements between the client with multiple network interfaces and the server machine.

The tool which will be a script could be deployed in two versions. One for client and the other for server. Based on the given parameters to the server version, it listens to the given TCP port and waits for an incoming socket connection from the client. Once the connection established between the client and the server, both machines could start capturing the packets by using command line packet analyzer tool such as **tcpdump**. The server could also run `ss` command simultaneously in order to investigate the established socket connection and extract the values such as *RTT*, *RTO*, *CWND*, *ssthresh* and etc. The capturing will continue during the connection

and will be written to a file on disk in a *csv* format when the connection is closed.

The direction along with the type of the traffic could be given to both versions. Hence, based on the direction of the traffic (i.e, download or upload) the server sends or receives the packets which will be generated based on the given type of traffic. For *bulk* traffic, the additional argument could be given to the script which specifies the amount of traffic that should be generated and transmitted from server to client in download direction and from client to server in upload direction. In this case, another file could be created by the script which contains the numbers of received segments from the socket connection in each time, preferably in epoch time format. This file could be used as Goodput values which represents the amount of received segments from the application point of view. For *onoff* and *stream* traffic, some existing tools could be used which generate the customized traffic based on the specified number of packets and the rate of transferring packets per second. Also the duration and the interval of the transfer could be specified in order to simulate the onoff traffic.

The congestion control algorithm could be another given argument to both scripts. Based on the direction of the traffic which species the sender, the congestion control algorithm could be changed on the sender via `sysctl` command based on the given name of the algorithm.

In addition, the interface name could be another argument for client script, in which based on the given interface name, the socket could binded to that interface resulting in establishing the connection to the server via the specified interface. Figure 3.1 illustrates the design of the measurement scripts.

The core of the experiments are the output files from the measurement scripts stated above. However, the trace files are usually too long due to the big number of captured segments they contain. Hence, some other scripts are required in order to filter the trace files and extract the needed parameters in a way that we could use them as a results of the experiment for comparison and analysis. This could be done by using some statistics on the distribution of the parameters values in order to obtain the mean and standard deviation and some other factors of each individual measurement and store the distribution of these samplings as another distribution (e.g, distribution of sample means from all experiments). In addition, some scripts could be implemented which draw the plot of the evolution of the parameters from beginning of the connection through its end which is described in following.

Plotting tool

The trace files resulted from measurement script need to be filtered in a way that only show the segments that have been transferred between the client

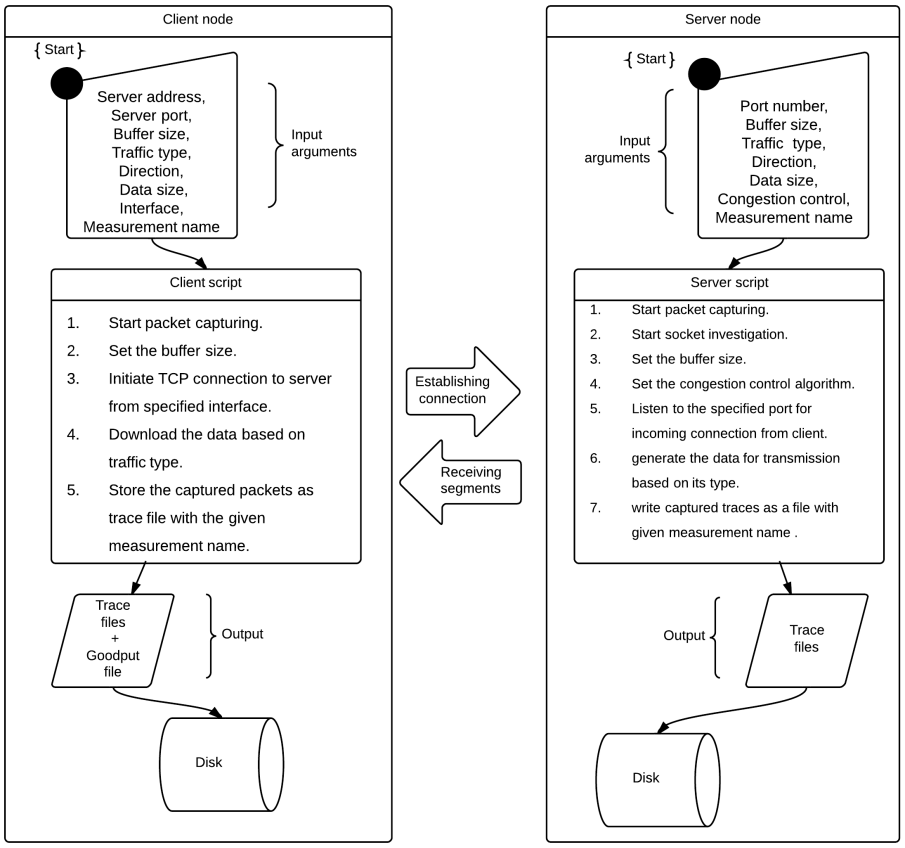


Figure 3.1: Measurement tool design, the client initiates the connection to the server and recieves the generated segments from the server.

and the specified port of the server (i.e, the other traffics destined and from other ports of the server must get excluded). This could be done by using a packet analyzer tool (e.g, Wireshark, Tshark, tcptrace and etc.) which provides the ability to filter segments to be shown based on the provided attributes.

After filtration, the needed parameters could get plotted by using some scripts and command line plotting tools. This visualization helps us better to observe the behavior of the key parameters that we're interested in each measurement. The plots that could be informative in each measurement could be as below:

- Throughput
- Goodput
- Congestion window (CWND)
- RTT
- Sequence number of the segments

The Goodput plot could be created from the Goodput file which is one of the output files from the measurement scripts. However, for Throughput plot the client trace file could be used since the amount of received segments at the interface is stored in the trace file. Hence, by using the packet analyzer tool the amount of received data (in Bytes or segments unit) in each time unit (e.g, epoch time) could be extracted and used.

For Congestion window, the output file resulted of the `ss` command which will be embeded in the server version of the measurement script could be used where the value of CWND in each time stamp is stored in that file. Moreover, for RTT and Sequence number the sender's (server in our case) trace file could be used. All these values could be extracted by combining the packet analyzer command with proper options. The X axis in all mentioned plots above could be *time* in seconds.

3.2 Procedure and Collecting data

The procedures and methods that should be considered for the experiment will be described in this section.

3.2.1 Repetition

In order to have a reliable results, the experiments should run for enough number of times, normally more than 30 measurements for each category in which we could have a valid sampling distributions for the QoS parameters that we're interested.

However, to avoid the *cache* problem which could exist on the routers on the path between two nodes and even in the client and server's network stack, there should be a quite reasonable interval between each measurement. In addition since we're interested in QoS differences with various congestion control algorithms between different networks, the results of measurements could be more realistic if the measurements take place in the times of the day in which the network is busy with reasonable amount of users (e.g, working hours).

Since the Mobile Broadband data plans are quite expensive comparing to other types of Internet connections (e.g, ADSL, Cable and etc.) and in order to not pass the quota limit of the MBB monthly data plan, the repetition must carefully considered in a way that both reliable results could be collected and fulfill the monthly data plan quota's terms and conditions. Therefore the amount of data that will be transferred should not be too big to eats the quota plan and not be too small which results in a not reliable measurement.

3.2.2 Expected output

As described earlier in 3.1.2, Two of the output files from the measurement tools are trace files generated by packet analyzing tool. The output format of the trace files which **tcpdump** generates are as following:

tcpdump output format

```
[source IP address].[source port] > [destination IP address].[destination port] \
[header flag] [initial sequence number]:[ending sequence number (implied)] \
([size in bytes]) [acknowledgement number] [advertised window size] ([other flags])
```

However, by using the packet analyzer tools (e.g, Wireshark, Tshark and tcptrace) again on the trace files, it is possible to apply the desired filters in order to extract the required information from them and write to the disk as **CSV** format, so they could later get used for plotting and statistics operations.

When the `ss` command executes with desired options, The output headers of the command is messy and needs to get properly arranged in

a CSV format. The following field headers of the desired output format for the values from `ss` command:

ss output format

```
timestamp,destination ip:port,congestion control,wscale,rto,rtt,mss,cwnd,ssthresh,\
rate
```

As described above, the expected output format of the files which will be used by plotting tool are as following:

- Throughput

  Throughput file format

  ```
  timestamp,received segment(s) size
  ```

- Goodput

  Goodput file format

  ```
  timestamp,segment(s) size received by application
  ```

- CWND

  CWND file format

  ```
  timestamp,congestion window (CWND) size
  ```

- RTT

  RTT file format

  ```
  timestamp,responded ACK's from the receiver rtt
  ```

- Sequence number

  Sequence number file format

  ```
  timestamp,sequence number of the sent segment(s)
  ```

3.2.3 Schedule of the experiments

By having the variants available for the experiments as described in 3.1, there should be a plan which highlights how the experiment should get executed. Since the main goal of this thesis is to find the differences of various TCP flavours in different MBB networks. Therefore the experiments should be planned as Algorithm 5.

For each buffer size as b and traffic type as t, the algorithm 5 produces a matrix consisting of c rows and l columns where each row represent one congestion control algorithm and each column represents each network. Hence, each element (i.e, a_{ij}), represents one experiment based on 3.1. For instance:

Algorithm 5 Experiments algorithm

Require: B, T, C, N
  for b in $(b \in B)$ **do** # buffer sizes
    for t in $(t \in T)$ **do** # traffic types
      $i \leftarrow 1$ # initiate first row
      for c in $(c \in C)$ **do** # congestion controls
        $j \leftarrow 1$ # initiate first column
        for l in $(l \in N)$ **do** # networks
          $a_{ij} \leftarrow Exp_{(b,t,c,l)}$ # set the elements of matrix
          $j \leftarrow j + 1$
        end for
        $i \leftarrow i + 1$
      end for
      $M_{b,t} \leftarrow$ Matrix of c rows and l columns with a_{ij} as elements
    end for
  end for

$$a_{13} \text{ in } M_{b_1 t_1} = Exp_{(B_1,T_1,C_1,N_3)}$$
$$And$$
$$a_{24} \text{ in } M_{b_2 t_3} = Exp_{(B_2,T_3,C_2,N_4)}$$

$$M_{b_1 t_1} = \begin{array}{c} \\ c_1 \\ c_2 \\ \vdots \\ c_C \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & ... & l_N \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{C1} & a_{C2} & \dots & a_{CN} \end{bmatrix} \end{array}$$

$$M_{b_1 t_2} = \begin{array}{c} \\ c_1 \\ c_2 \\ \vdots \\ c_C \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & ... & l_N \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{C1} & a_{C2} & \dots & a_{CN} \end{bmatrix} \end{array}$$

$$\vdots$$

$$M_{b_B t_T} = \begin{array}{c} \\ c_1 \\ c_2 \\ \vdots \\ c_C \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & ... & l_N \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{C1} & a_{C2} & \dots & a_{CN} \end{bmatrix} \end{array}$$

If we group each measurement as one of the described matrices (e.g, $M_{b_1 t_1}$) consisting $[a_{11} \dots a_{CN}]$ experiments, therefore the number of total measurement groups is as following:

$$\text{Total number of measurement groups} = \sum_{b=1}^{B} \sum_{t=1}^{T} M_{bt} \tag{3.2}$$

Also, the total number of experiments that should be executed for this thesis would be:

$$\text{Total number of experiments} = \sum_{b=1}^{B} b \times \sum_{t=1}^{T} t \times \sum_{i=1}^{C} \sum_{j=1}^{N} a_{ij} \tag{3.3}$$

In the other words, the measuring procedure in this thesis could be separated and grouped based on **buffer size** and **traffic type**, in which in each group, a set of experiments would run based on the elements of the group which consist of one **congestion control algorithm** per every available **network**. In this way, the behaviour of the congestion control algorithm could be observed on different available networks while the other parameters are same in each row of the measurement matrix and the next row runs the experiments with another congestion control algorithm again on every available networks. Hence, the behaviour of the congestion control algorithm could be reliably concluded based on all available variants.

3.2.4 Collected data

The collected data files which is the result of running experiment could be stored in a **Folder** hierarchy manner. Since Each matrix represents a measurement group, therefore the storing data method could be based on each measurement group.

This could be done by assigning the buffer size variation as the root of the folder hierarchy(b in matrix M) and setting the traffic type as the subfolder of the buffer size. Network would be a sub folder of traffic type and parent folder of the Congestion control algorithm folder. As noted, the network/congestion control/ which represent each experiment (e.g, a_{12} in matrix M) would exist for each sets of measurement group (i.e, matrix M). The general overview of the folder hierarchy for data collection in this thesis would be as following:

Buffer size/traffic type/network/congestion control/

A single measurement group (e.g, $M_{b_1 t_1}$) which consists of $a_{11} ... a_{CN}$ experiments could executed. Hence, the files resulting of measurement scripts and some other analysis scripts would be stored in the path specified above. For instance the files for experiment $Exp(B_1, T_2, C_1, N_4)$ could be stored as : $B_1/T_2/N_4/C_1/files$ In addition to the files, a folder which

holds plots of each individual experiment could be exist.

In order to have reliable and strong analysis each measurement group should get executed several times which logically results in having several Experiment with same attributes (e.g, $B_1/T_2/N_4/C_1/files$), which results in duplication or even overwritten. Therefore another level could be added under *congestioncontrol* as sub folder. The folder with a date and time of the experiment execution could make each experiment unique which consist its own files respectively.An example could be as following:

$$B_1/T_2/N_4/C_1/exp-date-time/files$$

3.3 Analysing data

In this chapter the methods that would take place in order to analyze the resulted data from experiments will be described.

3.3.1 Analysis tools

In order to analyze the results from each experiment and also summarize the repeated experiments and make a conclusion on them, another sets of scripts should be created which traverse through the result files for the required QoS parameters and do statistical actions such as calculating the mean and standard deviation of the distribution of the QoS values in every single experiments and store the statistic values in a file in *CSV* format so it could be used later.

Since each measurement matrix (M) would be run for several times, therefore there would be several individual experiments (e.g, a_{11}) which they have identified and distinguished by their unique suffix which is the combination of date and time appendix as described in previous section. However, another step could be defined in which , all the statistical values along with the experiment name for every single experiment could be extracted from the experiment folders and written in a summary file which contains the distribution of the means of each experiment. The format and location of the summary file containing distribution of the means and standard deviations of one specific QoS could be as following:

$$\begin{gathered} Exp(B_1,T_1,C_1,N_1)-date-time_1,mean,stddev \\ Exp(B_1,T_1,C_1,N_1)-date-time_2,mean,stddev \\ Exp(B_1,T_1,C_1,N_1)-date-time_3,mean,stddev \\ \vdots \\ Exp(B_1,T_1,C_1,N_1)-date-time_n,mean,stddev \end{gathered}$$

$$\text{Buffer size/traffic type/network/congestion control/} \tag{3.4}$$

3.3.2 Plots

By having enough number of repeated experiments (usually more than 30) in each measurement group and by having the distribution of the means of each individual experiment, the behavior of each congestion control (C) in every network (N) for each matrix (i.e, M_{bt}) could be easily visualised by using the script based plotting tools based on the summary files which located in 3.4. Statistical plots such as *boxplot, CDF (Cumulative Distribution Function), bar chart and scatter plots* could be good candidate to represent the results and help us to compare the differences with each other.

Chapter 4

Results

This chapter starts with describing the actual implementation of the measurements in which the materials and the parameters that have been considered for the measurements will be explained in detail. Following will be the result of all measurement groups which have been measured in this thesis and the data will be showed.

4.1 Implementation

In this section, the actual implementation will be described. The *Testbed* which have been used for running the measurement will be described, following by the actual design of the testbed. Furthermore, the operationalization of the measurement and the actual QoS parameters which needed to be observed will be described following by the functionality of the scripts.

4.1.1 Testbed

The experiments are running on **Nornet Edge**[1][21, 33, 46], which is a programmable testbed for measurements and network research. The Nornet Edge testbed is composed of customized single-board measurement nodes. The measurement node is connected to multiple MBB operators[2]. Figure 4.1 shows the NorNet Edge measurement node which is used as a client machine for running measurements in this thesis.

Three different MBB providers of Norway are considered for measurements. Therefore, the measurement node is connected to two 3G-UMTS operators by using similar USB 3G modems, and one 3G-CDMA2000 1xEV-DO Rev.A[3] network over USB. The 3G-UMTS modems support HSPA+ with theoretical data rate of up to 21.6 Mbit/s in downlink and 5.8 Mbit/s in uplink while the 3G-CDMA2000 modem supports a theoretical data rate

[1]NorNet: https://www.nntb.no/.

[2]The name of the operators will be kept anonymized.

[3]The CDMA modem is a standalone modem with LAN output which is connected to the node via LAN-to-USB adapter

Figure 4.1: NorNet Edge measurement node with USB 3G modems connected [46].

of 9.3 Mbit/s in downlink and 3.1 Mbit/s in uplink [26]. The actual measurement setup is illustrated in Figure 4.2. The location of the measurement node is in the Oslo region of Norway which is shown in Figure 4.3. The node has no mobility and is in static condition.

In addition another machine is used which has the role of orchestrating the measurements. This machine which is a desktop, runs the measurement scripts automated by *cron job* on both server and node through `ssh` command and collects the output files resulting of measurement scripts in order to extract the QoS parameters and furthermore plotting the results.

4.1.2 Measurements

The QoS parameters that are considered in this thesis are: *Goodput*, *RTT* and *One-way delay*. Also Three congestion control algorithms were considered: *TCP NewReno*, *TCP CUBIC* and *TCP Vegas*. NewReno is included in the measurement since it is a standardized algorithm and commonly is used as a baseline for comparison with other TCP congestion control algorithms. The TCP CUBIC algorithm is considered since it is the default congestion control algorithm in Linux and is widely deployed on Web servers [75]. TCP Vegas is included in a measurements since it is a representative of a delay-based congestion control algorithm while the two other algorithms are loss-based. The following command changes the congestion control algorithm to Vegas.

Setting Vegas as congestion control

```
$ sysctl -w net.ipv4.tcp_congestion_control=vegas
```

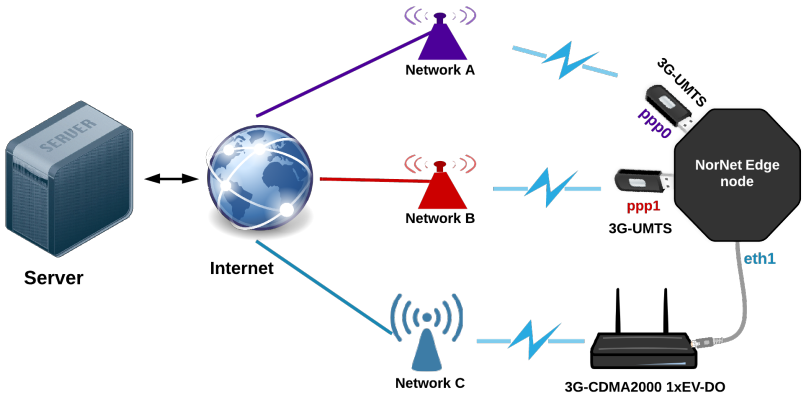


Figure 4.2: The overview of actual measurement setup.

Linux TCP were used for measurements with the default values for the parameters based on Table 2.2. However since we're only interested in congestion control algorithm and not the TCP flavor itself, therefore the congestion control algorithm will change by just changing the congestion control parameter according to Table 2.3. Although we referred to NewReno as a baseline congestion control which is commonly used, In practice, SACK-based recovery and FACK is used too.

In addition, the TCP metric cache is always flushed before establishing new TCP connection (i.e, beginning of each experiment). This can be done by the command below:

flushing the TCP metrics

```
$ sysctl -w net.ipv4.tcp_no_metrics_save=1
```

The considered traffic types for the measurements are: *Bulk*, *Onoff* and *Stream* traffics which are described in Section 3.1.1 on page 25. The different approach for traffic generation and the amount of sent segments for each traffic type will be described in detail in Section 4.2. *NetPerfMeter* [22] is used for traffic generation. NetPerfMeter is a network performance meter for the UDP, TCP, SCTP and DCCP transport protocols and has the ability to simultaneously generate bidirectional traffics and write the results as vector and scalar files.

For *Stream (application limited)* traffic type, two sub-types are considered. One is Streaming traffic type running with 50% of the average rate of each Exp elements e.g, a_{11} in bulk traffic type (Equation 4.1). The other Streaming traffic type experiments run with 25% of the average rate of their related elements in bulk traffic (Equation 4.2).

$$a_{cn} \equiv Exp^{Rate}_{(B_b,T_{str50},C_c,N_n)} = 0.5 \times Exp^{Rate}_{(B_b,T_{bulk},C_c,N_n)} \tag{4.1}$$

$$a_{cn} \equiv Exp^{Rate}_{(B_b,T_{str25},C_c,N_n)} = 0.25 \times Exp^{Rate}_{(B_b,T_{bulk},C_c,N_n)} \tag{4.2}$$

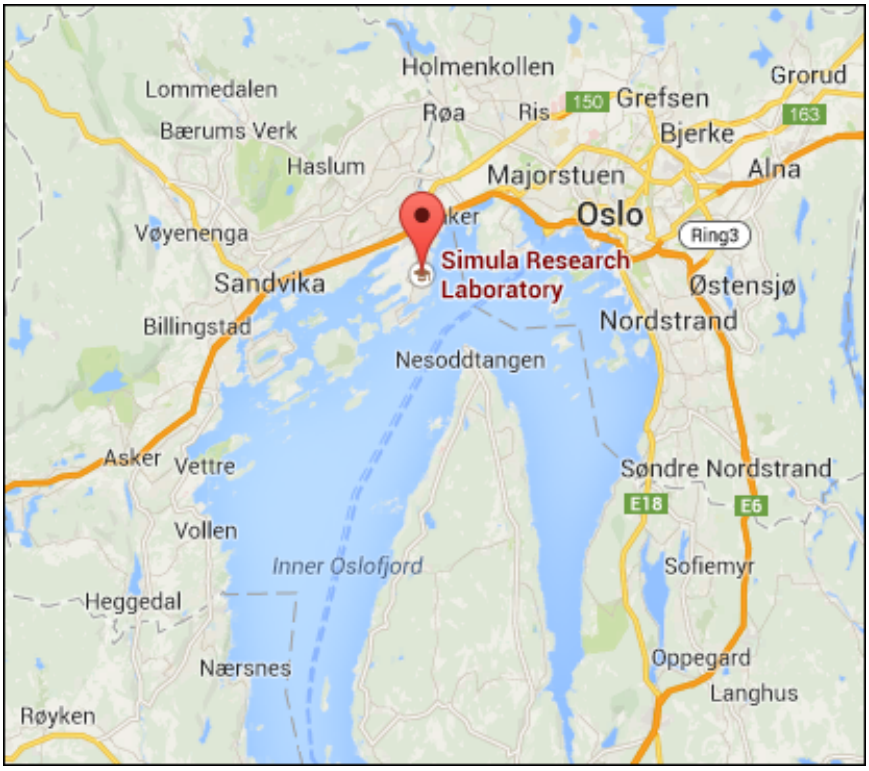


Figure 4.3: The location of the measurement node Oslo, Norway.

We are mainly interested in downlink direction. Therefore the direction of the traffic in the experiments are from Server to the node. Since the congestion control algorithm is a sender side implication, therefore the change of the congestion control algorithm takes place at server machine and not node. Also another parameter will be set in the way which starts every TCP connection in slow start phase. the command is as following:

Forcing the existing idle connections to start in slow start

```
$ sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

For buffer sizes, two parameters are changed: *send buffer size (tcp_wmem)* and *receive buffer size (tcp_rmem)*. Each parameters are vectors of three integers and can be configured via `sysctl` command in Linux. The format of the Linux TCP buffer size parameters are as following:

TCP send buffer size (wmem)

```
net.ipv4.tcp_wmem = min,default,max
```

TCP receive buffer size (rmem)

```
net.ipv4.tcp_rmem = min,default,max
```

In both parametrs, *min* value tells the kernel the minimum send/receive buffer size for each TCP connection, and this buffer is always allocated to a TCP socket, even under high pressure on the system. The *default* and *max*

values tells the kernel the default and maximum send/receive buffer sizes respectively.

As Described earlier in 3.1.1 on page 26, two sets of buffer sizes are considered for measurements: *Unlimited (unbounded)* buffer size which is equivalent to the amount of bytes that is considered for bulk traffic which can be considered a very large value compared to Linux default value. The *Limited (bounded)* buffer set size is considered based on the *Android* buffer set size operating in HSPA+ networks. Table 4.1 shows the value sets for each considered buffer sizes along with Linux default values for the node.

	Default value	**Limited (bounded) value**	**unlimited (unboounded) value**
tcp_rmem	4096, 87380, 507104	4096, 87380, 1220608	4096, 87380, 16777216
tcp_wmem	4096, 16384, 507104	4096, 16384, 1220608	4096, 16384, 16777216

Table 4.1: Send/Receive buffer size values

By having the variables required for the measurements, Equation 3.1 could be redefined as Equation 4.3. Hence, according to Algorithm 5, the measurement matrices (M) will be generated. Therefore, each measurement group (M_{bt}) which contains $C \times N$ experiments executed. The measurements groups are as following:

$$Exp_{(B,T,C,N)} \begin{cases} B \in \{\text{Limited, Unlimited}\} \\ T \in \{\text{Bulk, Onoff, Stream 50\%, Stream 25\%}\} \\ C \in \{\text{Reno, Cubic, Vegas}\} \\ N \in \{\text{eth1, ppp0, ppp1}\} \end{cases} \tag{4.3}$$

$$M_{b_{Unlimited},t_{Bulk}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

$$M_{b_{Unlimited},t_{Onoff}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

$$M_{b_{Unlimited},t_{str50}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

$$M_{b_{Unlimited},t_{str25}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

$$M_{b_{Limited},t_{Bulk}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

$$\vdots$$

$$M_{b_{Limited},t_{str25}} = \begin{matrix} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{matrix} \begin{matrix} \begin{matrix} l_{eth1} & l_{ppp0} & l_{ppp1} \end{matrix} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{matrix}$$

According to Equations 3.2 and 3.3, The total number of measurement groups and experiments are as following:

$$\begin{aligned} \text{Total number of measurement groups} &= \sum_{b=Unlimited}^{Limited} \times \sum_{t=bulk}^{str25} M_{bt} \\ &= 2 \times 4 \\ &= 8 \end{aligned}$$

$$\begin{aligned}\text{Total number of experiments} &= \sum_{b=Unlimited}^{Limited} b \times \sum_{t=bulk}^{str25} t \times \sum_{i=Reno}^{Vegas} \times \sum_{j=eth1}^{ppp1} a_{ij} \\ &= 2 \times 4 \times 3 \times 3 \\ &= 72\end{aligned}$$

The first measurement group, executed several times in downlink direction during workdays (Monday to Friday), between February and May of 2014. After having enough number of experiment samples in first group, the next measurement group execution took place and this execution method continues till the last measurement group execution. Two specific times of the day were considered for the execution of the measurements: 8:00 AM and 4:00 PM. Since there is a peak usage in the mornings from Mobile data users and less usage in the afternoon while the working hours is finished and most of the users around the measurement area are not existed. In addition, by providing this interval between each measurement, we can assume that the caching memory of the existed routers and other networking equipments in tha paths between the node and server are emptied and the measurement results are totally independent than each other. However, due to the expensive price of monthly data traffic for our MBB subscriptions, the samplings are limited to less than 20 samples for each experiment in each measurement group.

Once the execution of the measurement groups with Unlimited buffer size finished, The values for Limited buffer sizes in Table 4.1 were set manually on Server and node. After setting the buffer set size to limited, the measurement groups with limited buffer size were executed respectively. The following shows the commands used for setting the buffer sizes to our specified limited value.

setting TCP buffer size sets with limited values

```
$ sysctl -w net.ipv4.tcp_rmem=4096 87380 1220608

$ sysctl -w net.ipv4.tcp_wmem=4096 16384 1220608
```

Python and *Bash* were used as the language of the scripts that used in this thesis. The Python scripts were used for measurement scripts which executed at server and node and a mixture of Python and Bash scripts were used at the desktop machine for automating and scheduling the measurements, collecting output files from server and node, extracting the QoS values, creating statistic files from QoS values, creating summary files from all the available samples of each experiment and plotting.

tcpdump were used for capturing the sent and received packets and creating trace files on both server and node. Additionally, the `ss` tool used at the server which is the sender in our case to inspect the initiated socket connection and extract the TCP socket parameters such as CWND,

ssthresh, RTO and etc. *tcptrace* and *tshark* were used in order to filter the trace files and extract the I/O statistics such as RTT, sequence numbers, Throughput and Goodput. *R* programming language were used for generating statistical plots which takes the statistic summary files and creates the box and CDF plots.

In the following parts, the main functionality of the scripts along with their input arguments will be described.

Measurement scripts

In this part, the two scripts which were used for initiating the measurements between server and node is described. According to our considered design in 3.1.2, The two measurement scripts have implemented by using *Python* scripting language.

1. **Server Script**

   The considered input arguments for the server scripts is as below:

   Input arguments for server measurement script

   ```
   $ server.py [server ip] [server port] [data amount] [number of execution] \
   [start number] [direction: UL, DL] [filename] [congestion control] [traffic]
   ```

   The server script consists of two main functions: `tcpdump` and `ss` commands. Upon the execution of the script, The server creates a TCP socket which listens to the specified input port number. Then it sets the system settings through `sysctl` command based on the specified congestion control algorithm from input. However, the other setting such as flushing the metrics, buffer sizes (`tcp_rmem` and `tcp_wmem`) can be changed manually. The following shows the system setting that gets modified by executing the server script. The buffer size setting is adjusted for Unlimited size.

   ```
   1 subprocess.call('sysctl -w net.ipv4.
        tcp_congestion_control=' + str(cc_alg), shell=
        True)
   2 subprocess.call('ip tcp_metrics flush all', shell=
        True)
   3 subprocess.call('sysctl -w net.ipv4.
        tcp_no_metrics_save=1', shell=True)
   4 subprocess.call('sysctl -w net.ipv4.
        tcp_slow_start_after_idle=0', shell=True)
   5 subprocess.call("sysctl -w net.ipv4.tcp_rmem='4096
        87380 16777216'", shell=True)
   6 subprocess.call("sysctl -w net.ipv4.tcp_wmem='4096
        16384 16777216'", shell=True)
   ```

   Right after that, the server starts listening on all of its available interfaces for capturing sent and received packets by using `tcpdump` tool and writing the trace files od the captured packets. simultaneously, the socket inspection takes place by running `ss -into state`

established command as a daemon and the desired values extracted from its output and stored in a file as CSV format as the format expected in 3.2.2. Based on the specified traffic type, the server generates the traffic and send it to the node through the initiated TCP socket connection from the node to the server. For bulk traffic type, the string of random number of bytes equal to the *data amount* specified as input, generated and sent through the socket to the node. Once the total amount of the specified size has completely transferred, the script finishes writing to the files and terminates the tcpdump and ss commands. Hence, the output files - Trace file generated by tcpdump and ss file generated by ss command - are stored in the server's local disk. The Following is the example of execution of the server script with bulk traffic type and Reno as congestion control algorithm which generates 16 MB of data.

Execution of server script with Reno in bulk

```
$ server.py 128.39.37.182 40000 16 1 1 DL experiment1 reno bulk
```

The following is the first 5 lines of the ss file resulted from the execution of the server script.

```
1 1399356258.36,46.66.157.66:57788,ts,sack,reno,,
    wscale,5,5,rto,528,rtt,176/88,mss,1348,cwnd,10,
    ssthresh,,send,RATE,0.6127,retrans,,unacked,10,
    rcv_space,28960
2 1399356258.43,46.66.157.66:57788,ts,sack,reno,,
    wscale,5,5,rto,528,rtt,176/88,mss,1348,cwnd,10,
    ssthresh,,send,RATE,0.6127,retrans,,unacked,10,
    rcv_space,28960
3 1399356258.49,46.66.157.66:57788,ts,sack,reno,,
    wscale,5,5,rto,500,rtt,151/59,mss,1348,cwnd,14,
    ssthresh,,send,RATE,0.9998,retrans,,unacked,14,
    rcv_space,28960
4 1399356258.56,46.66.157.66:57788,ts,sack,reno,,
    wscale,5,5,rto,488,rtt,144/25,mss,1348,cwnd,21,
    ssthresh,,send,RATE,1.6,retrans,,unacked,20,
    rcv_space,28960
5 1399356258.62,46.66.157.66:57788,ts,sack,reno,,
    wscale,5,5,rto,436,rtt,93.5/18,mss,1348,cwnd,34,
    ssthresh,,send,RATE,3.9,retrans,,unacked,34,
    rcv_space,28960
```

For Onoff and Stream traffic where the rate of the traffic and sent time needed to be customized, NetPerfMeter was used. NetPerfMeter consists of two modes. *Passive mode* and *Active mode*. Running the NetPerfMeter by just providing the port number, puts it into the passive mode in which it listens to the port and waits for the connection to be initiated to the port from the machine which is running the NetPerfMeter in active mode. The following command puts the NetPerfMeter in background as Passive mode:

running NetPerfMeter in background as passive mode

```
$ netperfmeter port &
```

The Server in our case is running NetPerfMeter in passive mode and waits for the incoming connection from the node which is running in active mode. The NetPerfMeter tool provides the number of different options which could be used to generate complex and multi directional traffics. However the options that we're mainly interested to use for active mode are: choosing local address, inbound frame rate, inbound frame size, onoff feature, runtime, vector files. By providing these options to the NetPerfMeter command and execute it from the node, it goes in the active mode in which it receives the customized generated traffic from the server (passive node).

2. **Client script**

   The following shows the input arguments required for executing the client script:

Input arguments for client measurement script

```
$ client.py [server ip] [server port] [data amount] [number of execution] \
[start number] [direction: UL, DL] [filename] [interface] [traffic]
```

Since we're only interested in downlink measurements, therefore the server is always has a sender role while the node has the receiver role. According to this assumption we assigned the congestion control algorithm argument only for server script and not for the client (node) script, since it is only related to the sender. However, the buffer sizes are set in node at the beginning of the script similar to server.

The client script is similar to the server script in a way that it runs `tcpdump` and captures the packets. Since the node is equipped with multiple MBB interfaces, therefore the selection of the interface is based on the name of the interface as input of the script, in which the socket which will be used for initiating the connection to the server is binded to the specified interface. The following shows the example of the execution of client script which initiates the connection to the server described above from *eth1 i.e, Network C in Figure 4.2* in bulk traffic type and limited buffer size.

Execution of client script from eth1 in bulk

```
$ client.py 128.39.37.182 40000 16 1 1 DL experiment1 eth1 bulk
```

Once the client initiated the connection via eth1 to the server successfully, the 16 MB of random data generated by the server which has Reno set as its congestion control algorithm is sent back to back to the client. The following shows the output sample of the Goodput file resulted from the client script from this example.

Goodput file content

```
timestamp, Received Bytes
1399356258.391555,1348
1399356258.401692,1348
1399356258.402262,1348
1399356258.403144,1348
1399356258.40353,1348
...
```

The output above shows that at each timestamp, which represent miliseconds in each second, 1348 Bytes of data has been received by the socket. The trace files contain lots of detailed information about each segments and therefore their example output could not be shown here. However they were used by plotting scripts in which the QoS values will be extracted from them and the evolution plots were drown after each measurement.

Based on the design of the script in section 3.1.2, the Goodput file is one of the needed output files of the client script. For bulk traffic, this file could be created in a way that the size of each reception of the data by node from the socket in bytes is stored with the epoch timestamp of that moment in a CSV formatted file. Therefore this file could be referred as the goodput file, in which it represents the amount of data that the application receives at each timestamp without the additional protocol or other overheads in the amount of received data i.e, Throughput. However for Onoff and Stream traffic type, since the NetPerfMeter is used for traffic generation, it is not possible to interference with its socket to extract the goodput values, therefore the trace file of the node itself is used to create the goodput file based on the timestamps and the size of the each received segments.

Plotting scripts

After the successful execution of the measurement scripts on node and the server, the trace file and ss file resulted from the server and the trace file and goodput file resulted form the client script were generated. The steps described below are the actual procedures and scripts in which the QoS vales were extracted and the related plots were created.

- Filtering of trace files:

  The trace files must be filtered in order to only show the segments related to the initiated connection between the node and the server for the measurement i.e, sourced or destined from/to the specified server port e.g, 40000 in previously mentioned examples. The filtering script is bash script which uses `tcptrace` find the tcp stream number which is related to our specified port number and then filters the trace file by using the tcp stream number resulted from tcptrace as a filter expression for the `tshark` tool. The filtered trace file is written as a new

file.

```
1 tcptrace -n -f'(port==40000 or port==40001 or port
    ==40002 or port==40003 or port==40004) and
    data_bytes>1024' $tracefile
2 stream_line=$(head -1 ./PF)    #PF is the file which
    tcptrace creates and writes the stream number in
    it
3 tshark_stream=$((stream_line-1))
4 filter="tcp.stream==$tshark_stream"
5 tshark -r $tracefile -Y "$filter" -w
    $filtered_tracefile
```

- Congestion window plot

  A Python script used for plotting the Congestion window (CWND) and Slow Start threshold (ssthresh) in which it uses *Gnuplot* command line tool to draw the plots. The CWND and ssthresh values from the *ss* CSV file is used as Y axis values, while the X axis represents the duration in seconds. Figure 4.4, shows the CWND/ssthresh plot of the previously described measurement example.

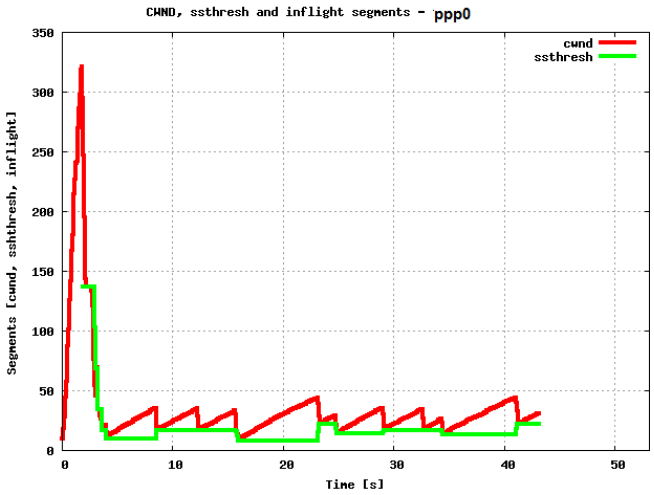


Figure 4.4: The evolution of *CWND* and *ssthresh* for Reno in bulk traffic at ppp0

- Sequence number plot

  For plotting the evolution of the sent segments sequence numbers, the filtered trace file of the server were used. the tshark command were executed on the filtered trace file but this time with another filter expression (i.e, frame time, sequence number) and the result were written to another CSV file as sequence number values in which this file used as the input for the sequence number plotting script which runs the draws the plot via Gnuplot with Y axis as sequence numbers and X axis as the time in seconds. The following is the first 6 rows

of the sequence number csv file. Also, Figure 4.5 shows the sequence number plot based on the this file which is related to the example described previously.

```
1 tshark -r $filtered_server_tracefile -T fields -e
    frame.time_relative -e tcp.seq > $seqnr_csvfile
```

Sequence number CSV file content

```
1 time,sequence number in [ /1024] format
2 0.000000000,0.0
3 0.000052000,0.0
4 0.177947000,2.6337890625
5 0.178732000,5.2666015625
6 0.178922000,7.8994140625
7 0.179272000,10.5322265625
8 ...
```

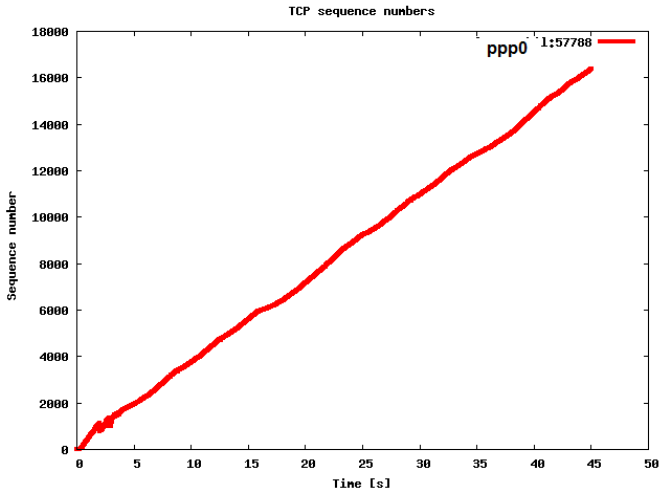


Figure 4.5: The evolution of *Sequence number* for Reno in bulk traffic at ppp0

- RTT plot

  For extracting and plotting RTT values, the same method as extracting sequence numbers were used, in which the filtered trace file of the server which is the sender is used since the RTT is the time it takes that the segment is sent and its related ACK is received. Therefore the tshark command is executed this time with another filter expression as following and the results is written to a csv file which again by using the Gnuplot the RTT plot is created with Y axis as RTT values and X axis is the time in seconds of the measurement. Figure 4.6, shows the plot of the evolution of the RTT for our example.

```
1 tshark -r $filtered_server_tracefile -T fields -e
    frame.time_relative -e tcp.analysis.ack_rtt >
    $rtt_csvfile
```

RTT CSV file content

```
timestamp,RTT
0.000000000
0.000052000,0.000052000
0.176139000,0.176087000
0.177673000
0.177947000
...
```

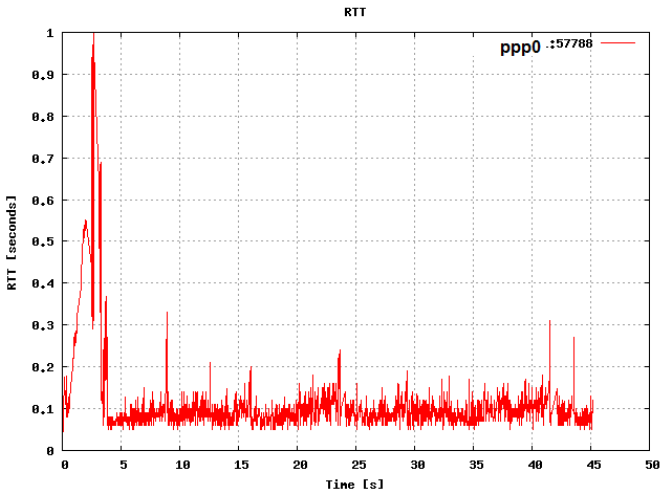


Figure 4.6: The evolution of *RTT* for Reno in bulk traffic at ppp0

- Throughput plot

  The Throughput plot were created by using the *i/o statistic* feature of tshark tool on the client's filtered trace file. However, the output of the command is not in a csv format, therefore the `STDOUT` of the executed tshark command were stored to a text file and the plot script parsed the text file and created the extracted values to another csv formatted file. Hence, the Throughput plot were created based on the file with its X axis as the time of the connection is seconds and the Y axis as the throughput value in each second. The following command extracts the Throughput amount received by the client with the 1 second interval. The output example of the csv file and the resulting plot is shown respectively.

```
tshark -q -z io,stat,1.0,"tcp.stream>=0" -r
    $filtered_client_tracefile > $throughput_STDOUT.
    txt
```

Throughput file content

```
seconds,received amount
1.0,0.07421875
2.0,0.140625
3.0,411.51953125
```

```
4.0,509.28125
5.0,527.828125
6.0,443.65234375
...
```

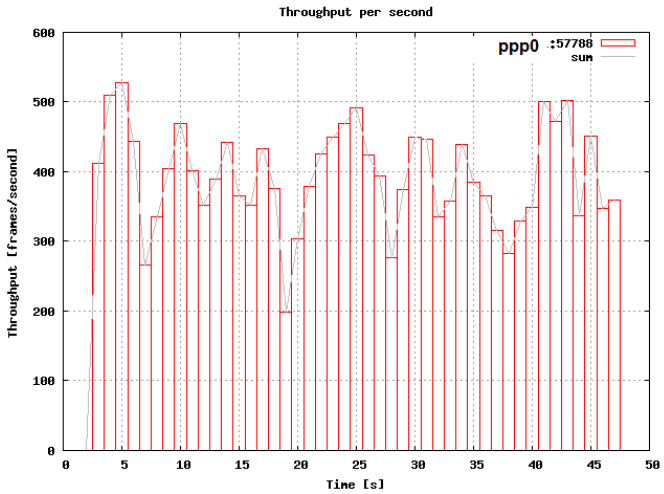


Figure 4.7: The *Throughput per second* for Reno in bulk traffic at ppp0

- Goodput plot

  By the execution of the goodput script plot which takes the value from the goodput file resulted from the client script described in page 46. The script summarize the amount of data received in each timestamp and plots the evolution of received goodput per each second. Also the average rate of goodput is calculated by dividing the total amount of received data in Kilo Bytes (KB) by the duration of the connection (measurement). Figure 4.8 shows the evolution of the goodput received in KB/S for our example.

  For Onoff and Stream traffic types where the NetPerfMeter is used for traffic generation, the goodput values were extracted from the NetPerfMeter vector log files. A simple Perl script used to parse the vector files for the Active node (client) and write the amount of received data in each timestamp similar to goodput file format created by client measurement script. Hence, the goodput plot of the Onoff and Stream traffic types were created too. The following is the example of the parsed vector file for the active node i.e, client.

NetPerfMeter active vector file

```
1 AbsolutTime,RelativeTime ,Action,AbsoluteBytes,RelativeBytes
2 000002,1399644017.54078,0.000000,0.000000,0.000,"Received",0,0
3 000008,1399644018.54157,1.000793,1.000793,3.712,"Received",120184,120184
4 000014,1399644019.54157,2.000793,1.000000,4.223,"Received",237472,117288
5 000020,1399644020.54151,3.000732,0.999939,3.354,"Received",357656,120184
6 000026,1399644021.54141,4.000639,0.999907,3.224,"Received",476392,118736
```

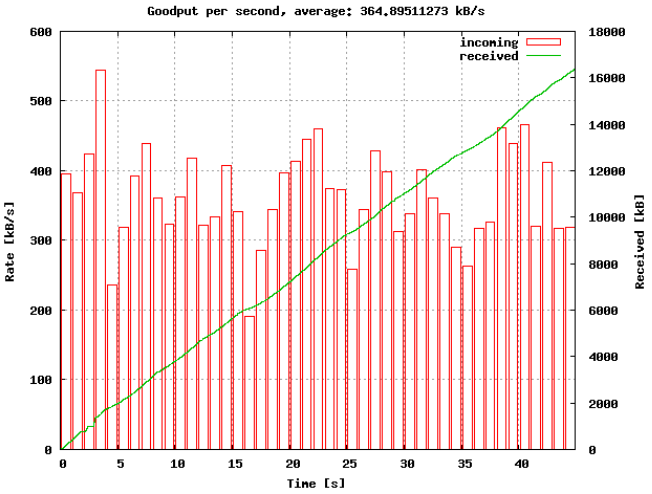


Figure 4.8: The *Goodput per second* for Reno in bulk traffic at ppp0

```
000032,1399644022.54148,5.000702,1.000063,3.780,"Received",595128,118736
...
```

The columns needed for creating the goodput file are the *RelativeTime* and *RelativeBytes*. Hence, the created goodput file is similar to the goodput file which created by the client measurement script. The following is the output of the parser script which creates the final goodput file.

Goodput file created by parser script from NetPerfMeter

```
timestamp,Received Bytes
1399644017.54078,0
1399644018.54157,120184
1399644019.54157,117288
1399644020.5415120184
1399644021.5414118736
1399644022.54148,118736
...
```

QoS calculation scripts

By having the files containing the distribution of the values such as *RTT* and *Goodput* which were resulted from the execution of the plotting scripts, it is needed to run statistics on the distribution of the mentioned values. Therefore another set of scripts used which reads the RTT and goodput files and calculates the mean and standard deviation of the values by using *numpy* module in Pyhton. The statistical values were written to another csv file with the name of the experiment.

The goodput values in the goodput file are related to each timestamp which represent miliseconds. However, for calculating the goodput per

second average, all the values within one second interval will be added together which represent the amount of data received in each second. Hence, the mean (average) were calculated based on these per second values.

Goodput per second values

```
Seconds,Goodput value
1,396.23828125
2,368.59375
3,650.3046875
4,377.80859375
5,251.43359375
6,325.15234375
...
```

For RTT, the values from the RTT field in RTT csv file which showed in Page 49 added to each other and divided to the total number of RTT values in that field to get the average RTT value. The following shows the content of the created files for *goodput per second* and *RTT* of the example described before.

Average Goodput

```
Experiment Name,goodput mean,stddev
bulk-ppp0-reno-nne679-Tue-06-May-08-03,368.653586648,71.5459417968
```

Average RTT

```
Experiment Name,mean RTT
bulk-ppp0-reno-nne679-Tue-06-May-08-03,0.113
```

Another considered QoS parameter were *One-way delay* i.e, application delay. It is the duration of time that it takes for the segments being sent to the moment that the receiver application receives it. In order to get the accurate value of this parameter, the timestamp option of the sent segment could be enabled. Hence, the sender labels each segment being sent with the timestamp of the sent moment. Once the application receives the segment, it can extract the one-way delay value by getting the difference between the sent timestamp and current timestamp since they both are using epoch time format. However, in order to accurately get this value, the time value on both machines should be exactly synced.

This can be done by using the Network Time Protocol (NTP). Before each measurement, the NTP on both server and node could be resynchronized. But since the accuracy of the NTP is not better than 10 or 20 ms for WAN networks [4] and is dependent to the machine's hardware, and by the assumption that one way delay value could be less than 10 to 20 ms, another workaround were needed which described in [4].

Due to the limitations in our implementation, we couldn't apply this method for acquiring one-way delay. Instead, we used the difference between each timestamp in our goodput files which represents the amount of time that it took for the application to receive the next segment.

Although this value is the difference of the goodput timestamps and not the real absolute time that took for the segments to received by the application, we can refer to this value as a representative of pattern that application receives the segments and can compare this pattern through our different variables i.e, networks, congestion controls, etc. Hence, we call this value as one-way delay (application delay) throughout this report.

The following shows the first 6 lines of the one-way delay values calculated form the goodput file described earlier.

One-way delay values

```
timestamp differences
0.010136842727661133
0.0005700588226318359
0.0008819103240966797
0.0003859996795654297
0.017000198364257812
0.00045490264892578125
...
```

Similar to RTT and Goodput average values, the mean (average) of these one-way delay values were calculated and were written to another file with in csv format as following.

Average one-way delay file content

```
Experiment name,One-way delay mean,stddev
bulk-ppp0-reno-nne679-Tue-06-May-08-03,0.0038148321129,0.00912750750508
```

The methodology for calculating one-way delay for Onoff and Stream traffics where NetPerfMeter is used for traffic generation is different. As illustrated earlier in Page 51, the vector file resulted from the NetPerfMeter which used to create Goddput file, shows the total amount of received bytes in every one second in each line. Hence, the one-way delay calculation which calculates the difference of each two lines in the goodput file would result in same value (approximately 1 second). Therefore, the method which were used to get the One-way delay value in bulk traffic type could not be used for these traffics.

The only possible way to calculate the one-way delay in our setup were to extract this information from the client's (node) trace file. This were done by using the tshark filter expression which displays the timestamps in the epoch format of the received segments with server as their sources. The results of this command were written to another file in which the one-way delay calculator script takes the differences between each of the rows i.e, timestamps and generates the one-way delay values file similar to bulk traffic and furthermore the mean of the distribution of the values were calculated and were written to a file similar to bulk traffic.

The following shows the tshark command which were used to generate the timestamp values of the segments.

```
1 tshark -t e -r $filtered_client_tracefile -Y "ip.src
    ==128.39.37.182" -T fields -e frame.time
```

As a result the file which contains all the timestamps in which the segments have been received were created. This file expands each seconds of the illustrated goodput file created by parser script from NetPerfMeter in Page 52 in milliseconds. The following shows the first 6 rows of the created file containing the extracted timestamps from filtered trace client file.

file containing epoch timestamps for each received segments

```
1 1399644018.010442000
2 1399644018.020286000
3 1399644018.033289000
4 1399644018.045291000
5 1399644018.056289000
6 1399644018.064437000
7 ...
```

As can be seen from above, the lines in this file is the expansion of each timestamp (in milliseconds) which the segments were received. These 6 lines are the expansion of the second row (18th second) in file showed in Page 52. Furthermore, the one-way delay calculator script, takes the differences between each rows in this timestamp file and similar to bulk traffic type, a file contains all the difference values and a file which contains the mean of the values along with the experiment creted respectively. The following shows the average one-way delay file content for Stream traffic i.e, generated by NetPerfMeter.

Average one-way delay file content for stream 50% traffic

```
experiment name, one-way delay mean,stddev
str50-eth1-reno-nne679-Fri-09-May-16-00,0.0121915651541,0.0053583169539
```

In addition to the described QoS parameters, another value which was interesting to observe for each individual experiment was number of changes in *ssthresh* value. As described in background section, Page 7, by the occurrence of packet loss or timeout, the ssthresh value changes based on the cwnd value. Although this behaviour only implies for loss based congestion control algorithms (i.e, Reno and Cubic), this value could help in analysis and comparison between mentioned loss based congestion controls by referring to the average of this value for each measurement group.

The *ssthresh change* value were extracted from the *ss* file by observing the number of times where the value of *ssthresh* is changed. Similar to previously mentioned QoS calculations, this number were written to a file along with experiment name in csv format as following example.

Number of ssthresh cahnges file

```
Experiment name,Number of ssthresh changes
bulk-ppp0-reno-nne679-Tue-06-May-08-03,13
```

In the shown example above, the *ssthresh change* value here indicates that the *ssthresh* has changed 13 times, which by looking at Figure 4.4 we can see the exact number of times that the ssthresh i.e, the green line has changed which in this experiment they are all due to the packet loss.

Orchestration script

As mentioned in Section 4.1.1, a desktop machine was used in order to run the measurements scripts simultaneously on the server and node via `ssh` connection. A bash script named *exp*, created on this desktop machine executed by the provided argument in which based on the arguments such as server address, server port, traffic type, congestion control, network and etc. it executes the measurement scripts on both server and node.

Each execution of the *exp* script represents on Experiment i,e. $Exp(B, T, C, N)$. Since each experiment were meant to be repeated for several times, Therefore, the experiment name which this script provides for the measurement scripts should be unique and is based on the scheme described in Section 3.2.4, which is traffic type-network-congestion control-node name-date-time.

Once the measurements finished, the script collects the resulted files from both measurement machines in order to run the plotting script which is another bash script that automates the execution of the plotting scripts and QoS calculation scripts.

Once the plotting and QoS calculation scripts executed, the *exp* script, creates the path related to its received input arguments as equation described in 3.2.4. It also reads the contents of each QoS average values from their created stat files and append them as a new line in the summary files described in Section 3.3.1 for each QoS parameter for later statistical analysis.

Additional bash script was used which automates the execution of the *exp* script in a way that one measurement group of experiments e.g, $M_{b_{limited},t_{bulk}}$ executed. This script uses two-level *for* loop which iterates with the congestion control algorithms as first level and network providers as its second level loop. The sleeping time between each iteration in the network providers loop is 120 seconds and 900 seconds for the congestion control loop.

The following shows the loops where automates the execution of the experiments and completes one measurement group ($M_{b_{limited},t_{bulk}}$) and creates the folder hierarchy inside *cron-exp* folder matrix described in Page 62. The buffer size were set as limited size manually before the execution of

the script.

```
for cc in reno cubic vegas
do
    echo
    PORT=40000
    for int in eth1 ppp0 ppp1
    do
        echo 'Running Command with $cc as Congestion
           Control and $int as interface'
        ./exp.sh 128.39.37.182 $PORT 16 1 1 DL $int $cc
           cron-exp bulk TCP nne679
        echo
        sleep 120
        ((PORT++))
        echo
    done
    sleep 900
done
```

Finally the execution of the automation script were added as a *cron job* with the schedule described in Page 43. After having enough number of experiments for each measurement group the traffic type and buffer size changed until all the measurement groups were executed and had enough number of repeated experiments.

Analysis plots

By finishing the measurements, the analysis plotting needed to be done in order to summarize the behaviour of the different congestion control algorithms in different networks based on the existed QoS parameters.

The following is one example of the summary file contains multiple lines of different repeated experiments along with their average and stddev values for *Goodput per second* as QoS in *eth1* network with *Reno* as its congestion control algorithm and with *bulk* traffic type and *unlimited* buffer size.

Goodput per second summary file for $M_{b_{unlimited},t_{bulk},c_{Reno},n_{eth1}}$

```
Experiment Name,Average goodput per second,stddev
bulk-eth1-reno-nne679-Fri-04-Apr-16-00,165.748325893,25.7972994805
bulk-eth1-reno-nne679-Mon-07-Apr-08-00,195.719879518,28.9097729184
bulk-eth1-reno-nne679-Mon-07-Apr-16-00,144.822524889,23.9795301902
bulk-eth1-reno-nne679-Tue-08-Apr-08-00,196.639871988,36.2611144664
bulk-eth1-reno-nne679-Tue-08-Apr-16-00,173.237699468,23.124142038
bulk-eth1-reno-nne679-Wed-09-Apr-08-00,199.572503811,19.4536403654
bulk-eth1-reno-nne679-Wed-09-Apr-16-00,194.096912202,27.5099317738
...
```

As described earlier in Section 3.3.1, The place of the summary files are as Equation 3.4. For instance, in the example above, the place of average goodput per second summary file is at:

unlimited/bulk/eth1/reno/

For every 9 experiments i.e, elements in one measurement matrix, there were four summary files as below:

- Average Goodput per second
- Average RTT
- Average One-way delay
- number of changes in ssthresh value

The **R** programming language was used in form of scripts in order to traverse through summary files for each QoS in every experiments folders and add the values as a data set so it could create statistical plots from the distribution of the imported data sets.

4.2 Measurement results

In this section the actual result of the experiments based on their measurement group will be shown .

4.2.1 Bulk traffic

In this section, the results from the measurement groups with *bulk* as traffic type with both limited and unlimited buffer set sizes will be shown.

For this type of traffic, 16 MB of data is sent from the server to the node. The MSS (Maximum Segment Size) is 1448 Bytes, which means that each packet size contains 1448 Bytes of data. The sending rate is not limited and server tries to send as much data as possible i.e, saturates the network.

Figure 4.9, is the sample throughput plot of one experiment with bulk traffic type which shows how server sends the segments.

Unlimited buffer size

The results in this section represents below measurement group matrix. Figure 4.10, shows the Boxplots which are created based on the summary files described in Section 4.1.2. The buffer size values were set based on the values mentioned in the *Unlimited* field of the Table 4.1.

$$M_{b_{Unlimited},t_{Bulk}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

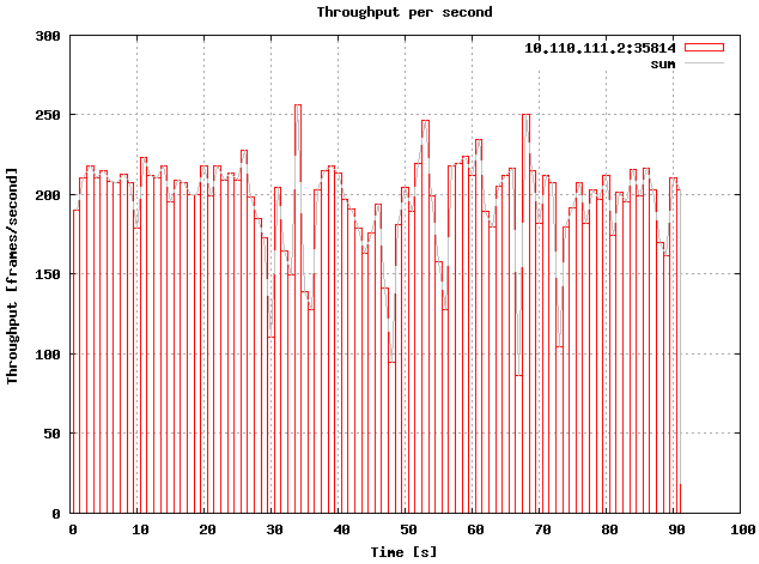


Figure 4.9: The sample of throughput pattern in bulk traffic type

Limited buffer size

Figure 4.11, shows the results from the similar experiments resulted previously , but this time with limited buffer set sizes which are based form the *limited* field of the Table 4.1.

The measurement group matrix which represents these experiments is as following:

$$M_{b_{Limited},t_{Bulk}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

4.2.2 Onoff traffic

In order to generate Onoff traffic, the NetPerfMeter tool were used as described on Page 45. Upon the execution of the client measurement script, the NetPerfMeter command were executed as following:

```
1 netperfmeter "+str(server_addr)+":"+str(server_port)+" -
    control-over-tcp -local="+str(ip)+" -vector="+
    filename+".vec.bz2 -scalar="+filename+".sca.bz2 -tcp
    const0:const0:const0:const1400:description='onoff':
    onoff=0,+1,+5,+1,+5,+1,+5,+1,+5,+1,+5,+1,+5,+1,+5,+1
       ,+5,+1,+5,+1 -runtime=60
```

The parameters were used in this command makes NetPerfMeter's passive node i.e, server to start sending packets with 1400 bytes set as MSS value. The sending frame rate is unlimited, meaning that the server tries to send as much segment as is possible based on its cwnd size per each seconds. After initiation of the connection, the server starts sending segments for 1 second and then stays idle for 5 seconds. At 6th second, again the server starts sending for 1 second and stays idle for the next 5 seconds. This pattern continues until the specified connection duration exceeded which is at second 60th.

Figure 4.12, shows the sample throughput plot in which the server sends traffic for one second and stays idle for five seconds in connection duration.

Unlimited buffer size

The results in this section represent the following measurement group matrix.

$$M_{b_{Unlimited},t_{onoff}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

Figure 4.13 shows the boxplots of QoS's considered in Onoff traffic type with limited buffer size which is based on Table 4.1.

Limited buffer size

Figure 4.14, shows the boxplots resulted from experiments in following measurement group in which the buffer set size are limited values according to Table 4.1. The following is the measurement group matrix representing these series of experiments.

$$M_{b_{Limited},t_{onoff}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

4.2.3 Stream traffic

In order to run the measurements with Stream traffic type, the goodput average values in each experiment i.e, a_{ij} elements were used as baseline goodput values in each buffer set sizes. These baseline values together with MSS value (1448 Bytes) and data size (16 MB) were hard-coded in an script in which it calculates the desired output rate by dividing the specified percentage (from input as argument) of the baseline rate values to the MSS size. In addition, the number of packets were calculated by dividing the data size in Bytes unit to the MSS size. Finally, the runtime were calculated by dividing the number of packets to the output rate. The Perl script code which does the described calculation is as following.

```
1 my $fullrate = $rate{$cc}{$int};
2 $outrate = ceil((($fullrate * ($percent / 100)) * 1024) /
      $mss);
3 # calculate the packet numbers in the size
4 my $packets = (($size * 1024) * 1024) / $mss;
5 my $runtime = ceil($packets / $outrate);
```

The client measurement and orchestration scripts were reconfigured to accept *output rate*, *mss* and *runtime* values as additional input arguments. The client measurement script executes the NetperfMeter command with specified frame rate and frame size and runtime as following.

```
1 netperfmeter "+str(server_addr)+":"+str(server_port)+" -
    control-over-tcp -local="+str(ip)+" -vector="+
    filename+".vec.bz2 -scalar="+filename+".sca.bz2 -tcp
    const0:const0:const"+str(rate)+":const"+str(mss)+":
    description='stream':maxmsgsize="+str(mss)+" -runtime
    ="+str(runtime)
```

The average Goodput values from bulk traffic type which were used as baseline for stream traffic type are shown in Table 4.2 and 4.3 for Unlimited and Limited buffer set size respectively.

	eth1	**ppp0**	**ppp1**
Reno	181.405	304.741	1137.788
Cubic	188.45	352.578	1273.132
Vegas	96.747	155.148	237.16

Table 4.2: The average Goodput per second in KB/s values in bulk traffic with **Unlimited** buffer set sizes

Figure 4.15 and 4.16 show the throughput plots for stream traffic with 50% and 25% rate of the bulk traffic type which showed in Figure 4.9.

	eth1	**ppp0**	**ppp1**
Reno	252.571	359.85	1120.836
Cubic	270.437	361.136	1177.917
Vegas	91.079	134.56	220.186

Table 4.3: The average Goodput per second in KB/s values in bulk traffic with **Limited** buffer set sizes

Stream with Unlimited buffer size

Figures 4.17 and 4.18 show the boxplots resulted from the measurement group matrices as following.

$$M_{b_{Unlimited},t_{str50}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

$$M_{b_{Unlimited},t_{str25}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

Since the Goodput averages were set according to the baseline values specified in Table 4.2, the Goodput values for stream traffic are constant and similar to 50% and 25% of the goodput values in bulk traffic. Hence, the goodput boxplots are similar to bulk traffic and were omitted from the Figures shown in this section.

Stream with Limited buffer size

Similar to the previous sections, in this section the boxplots which represent the last two measurement groups are shown. Figures 4.17 and 4.20 show the results from the executed experiments with Stream traffic type with 50% and 25% of the bulk traffic's Goodput per second rate. The followings are the measurement group matrices which their results are presented in his section.

$$M_{b_{Limited},t_{str50}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

$$M_{b_{Limited},t_{str25}} = \begin{array}{c} \\ c_{Reno} \\ c_{Cubic} \\ c_{Vegas} \end{array} \begin{array}{c} \begin{array}{ccc} l_{eth1} & l_{ppp0} & l_{ppp1} \end{array} \\ \begin{bmatrix} a_{(Reno,eth1)} & a_{(Reno,ppp0)} & a_{(Reno,ppp1)} \\ a_{(Cubic,eth1)} & a_{(Cubic,ppp0)} & a_{(Cubic,ppp1)} \\ a_{(Vegas,eth1)} & a_{(Vegas,ppp0)} & a_{(Vegas,ppp1)} \end{bmatrix} \end{array}$$

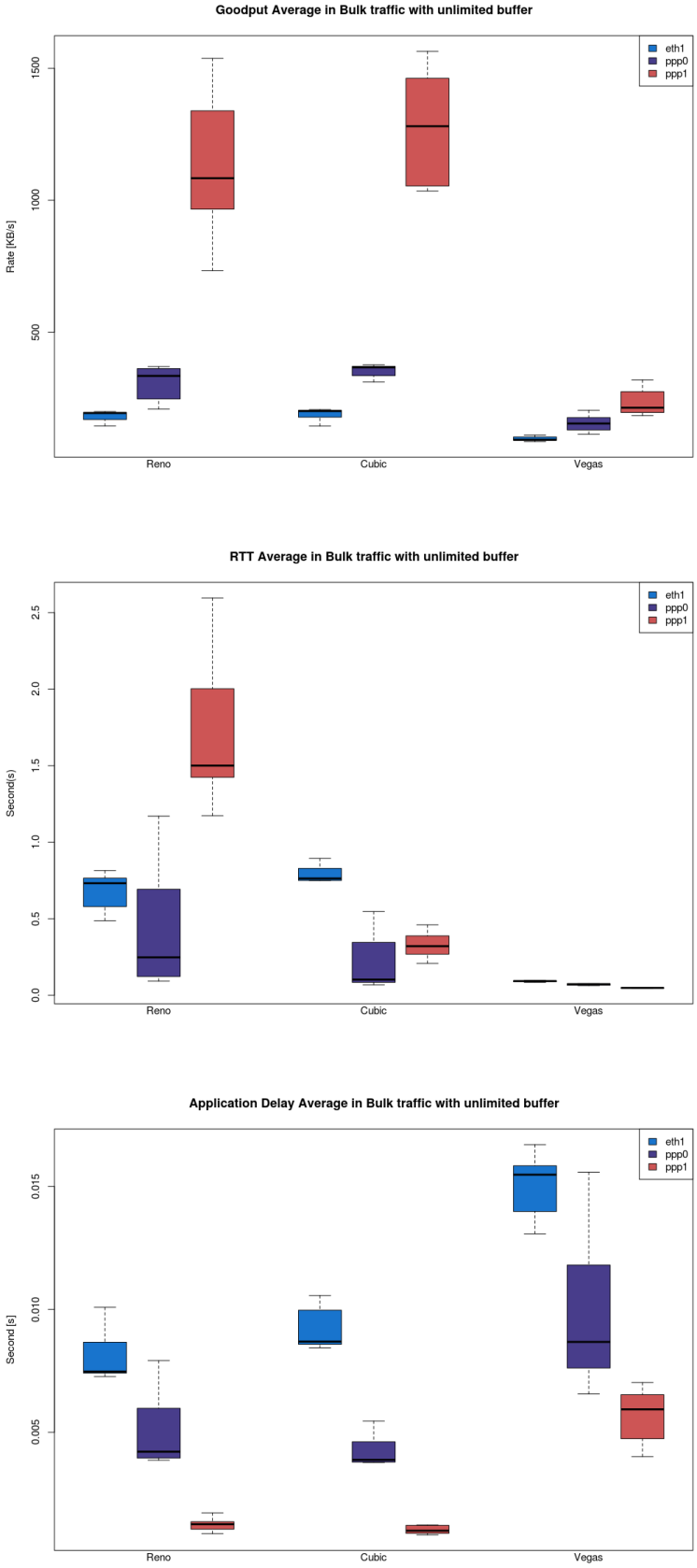


Figure 4.10: Bulk traffic with unlimited buffer sizes

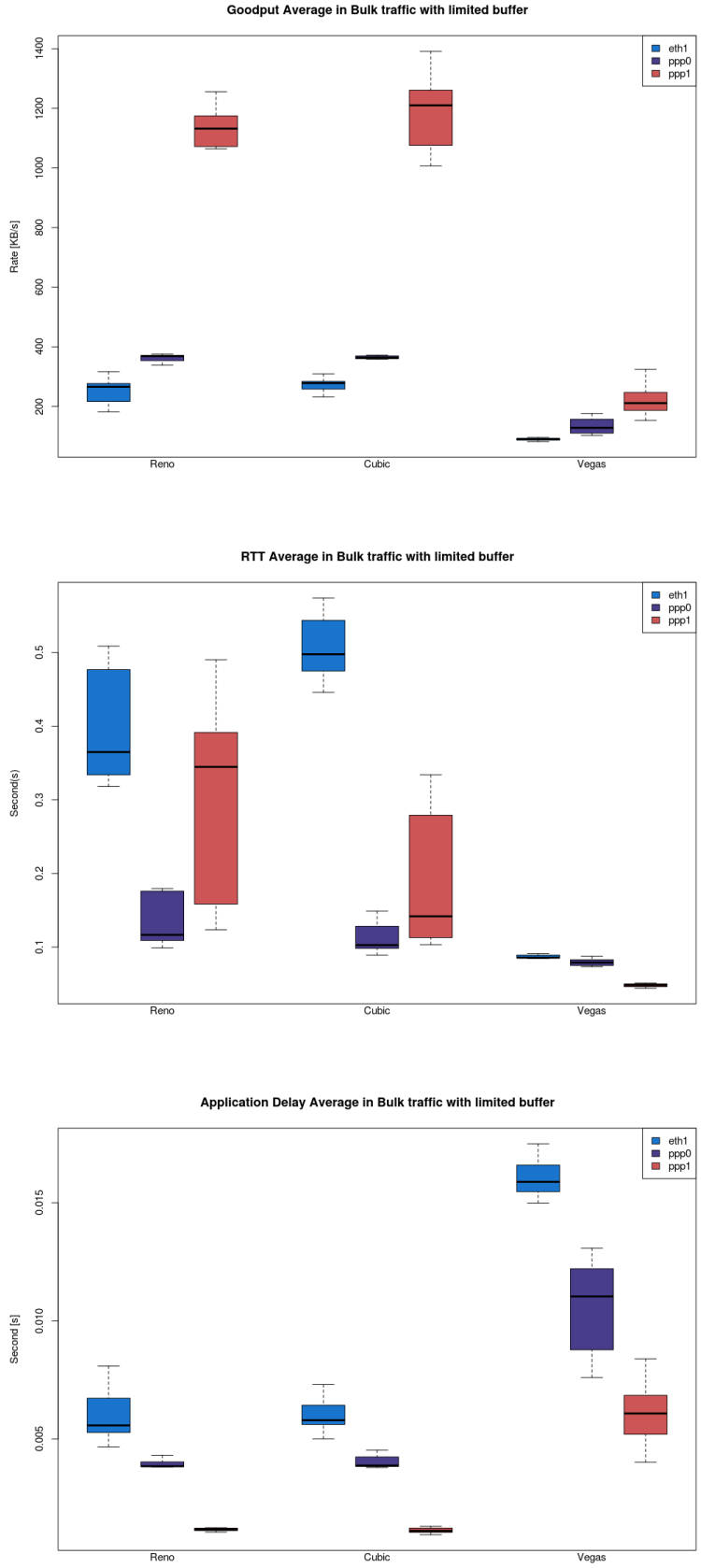


Figure 4.11: Bulk traffic with limited buffer set sizes

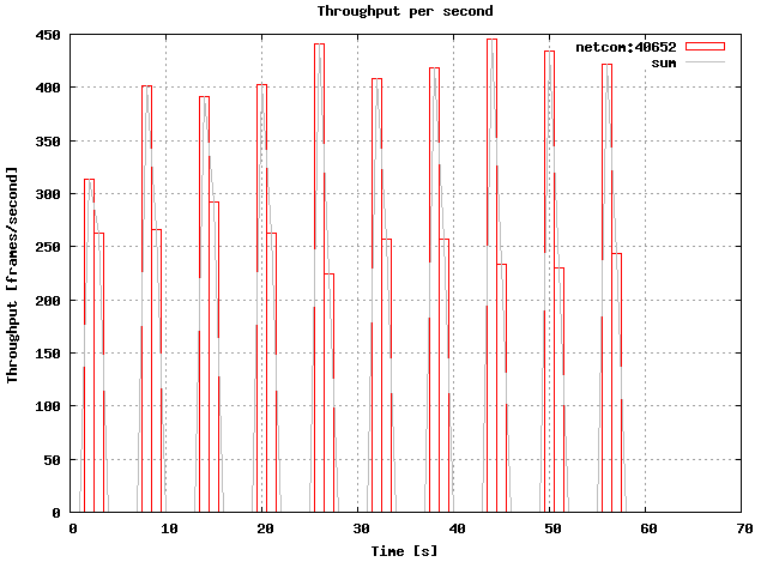


Figure 4.12: The sample of throughput pattern in Onoff traffic type

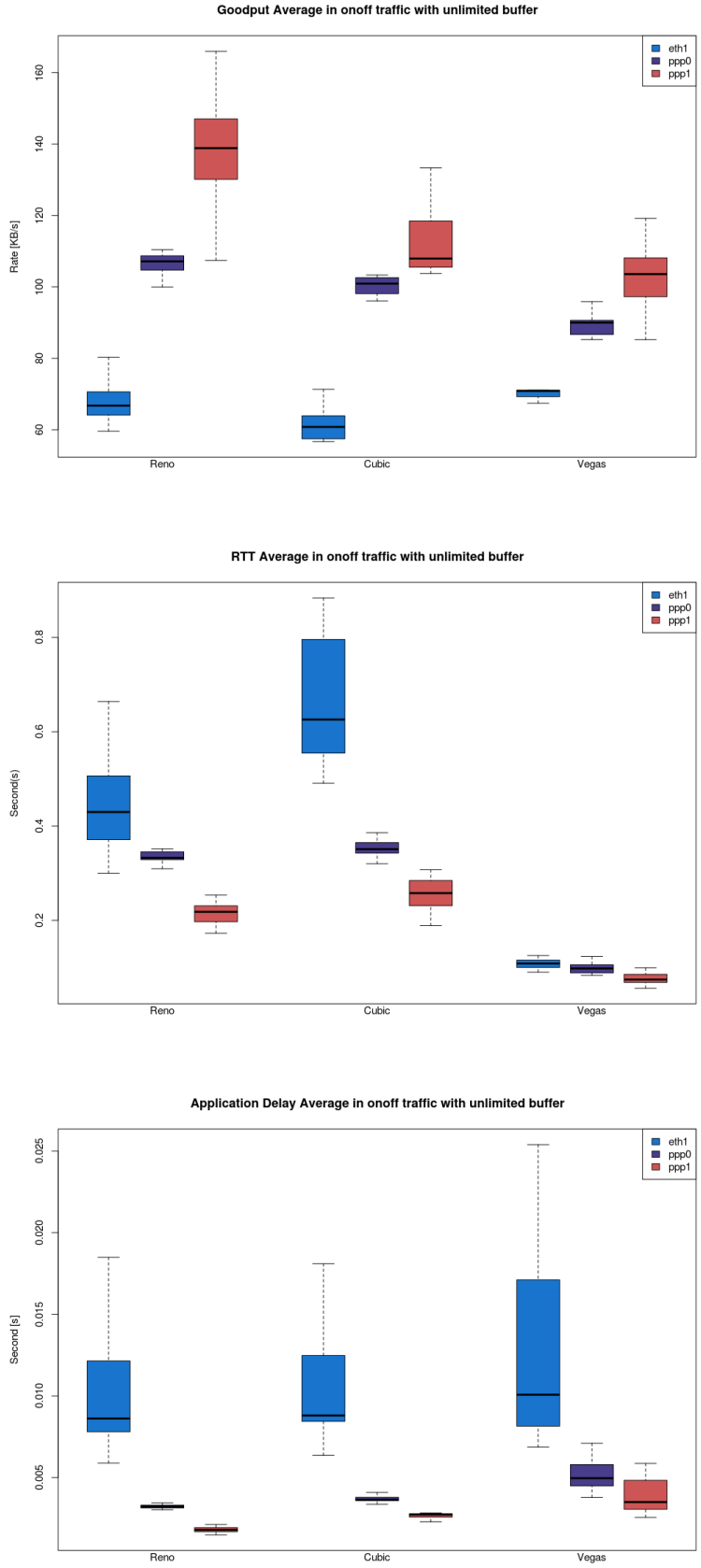


Figure 4.13: Onoff traffic with unlimited buffer sizes

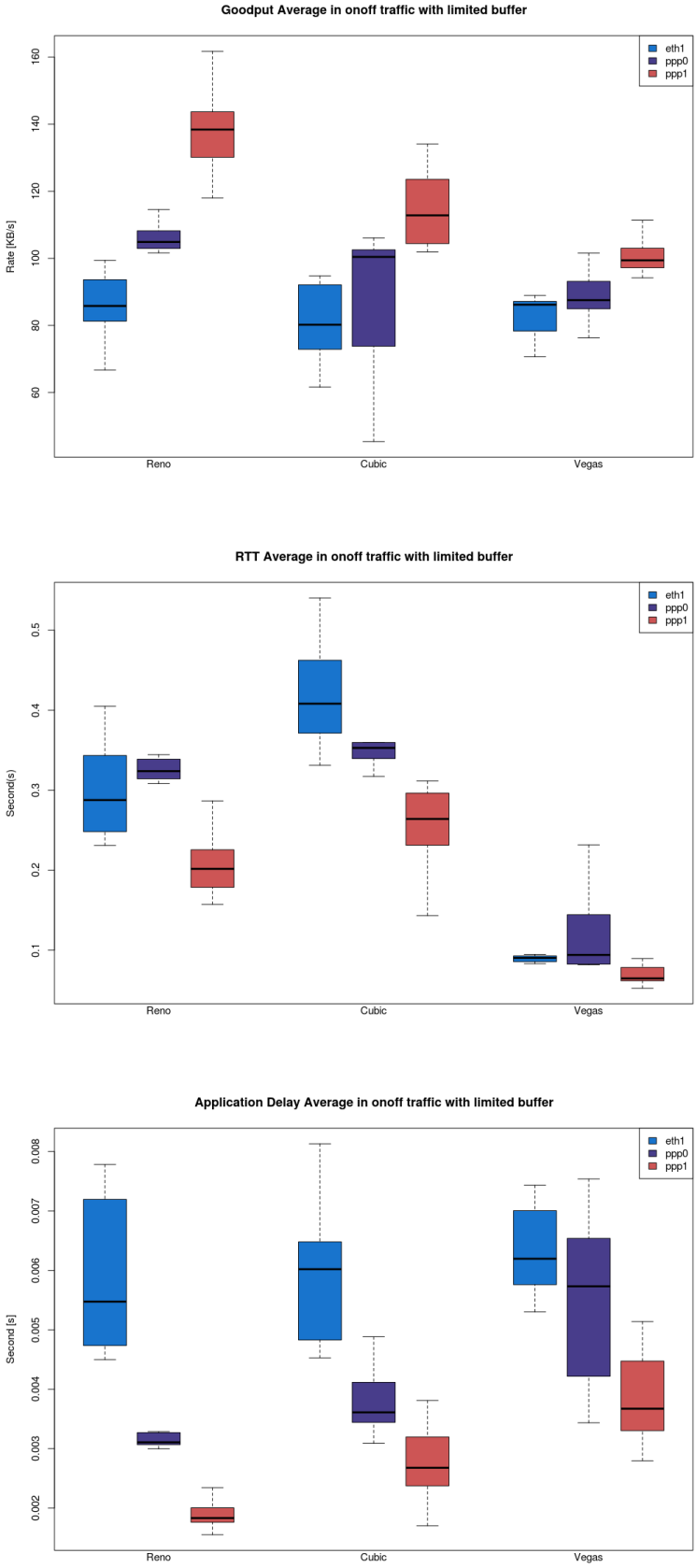


Figure 4.14: Onoff traffic with limited buffer sizes

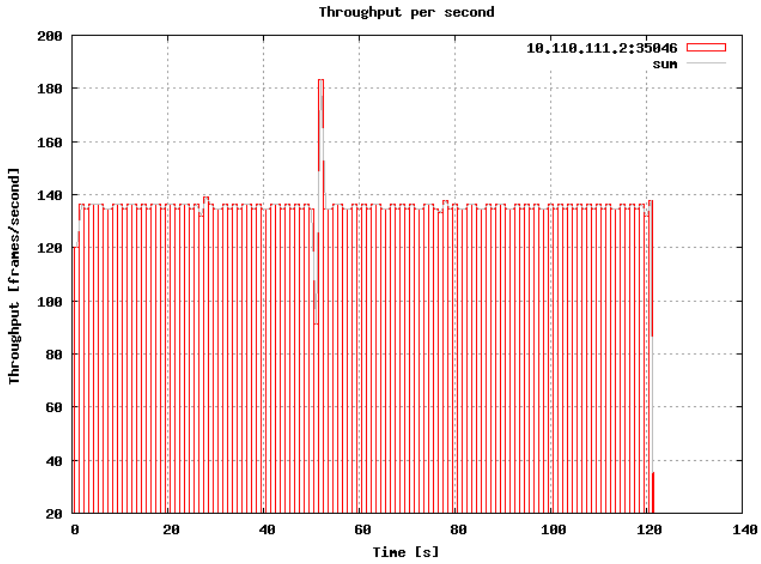


Figure 4.15: The sample of throughput pattern in stream traffic type with 50% of the bulk rate

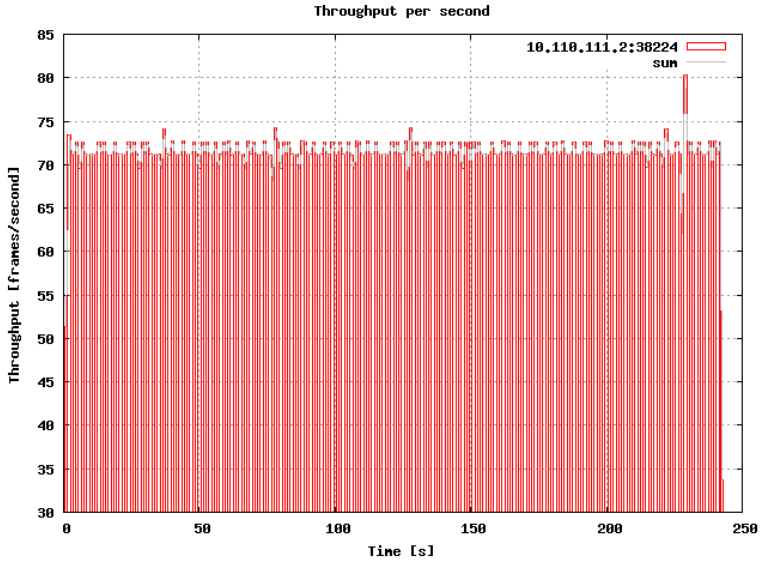


Figure 4.16: The sample of throughput pattern in stream traffic type with 25% of the bulk rate

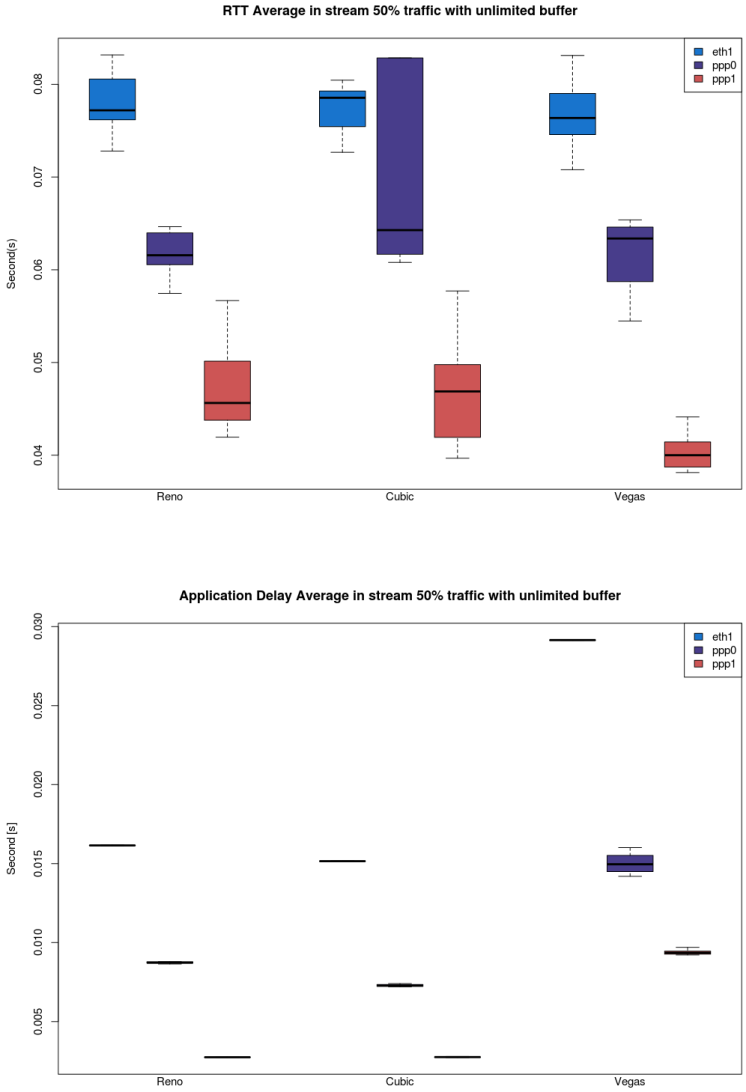


Figure 4.17: **Stream** traffic with 50% of bulk goodput rate unlimited buffer sizes

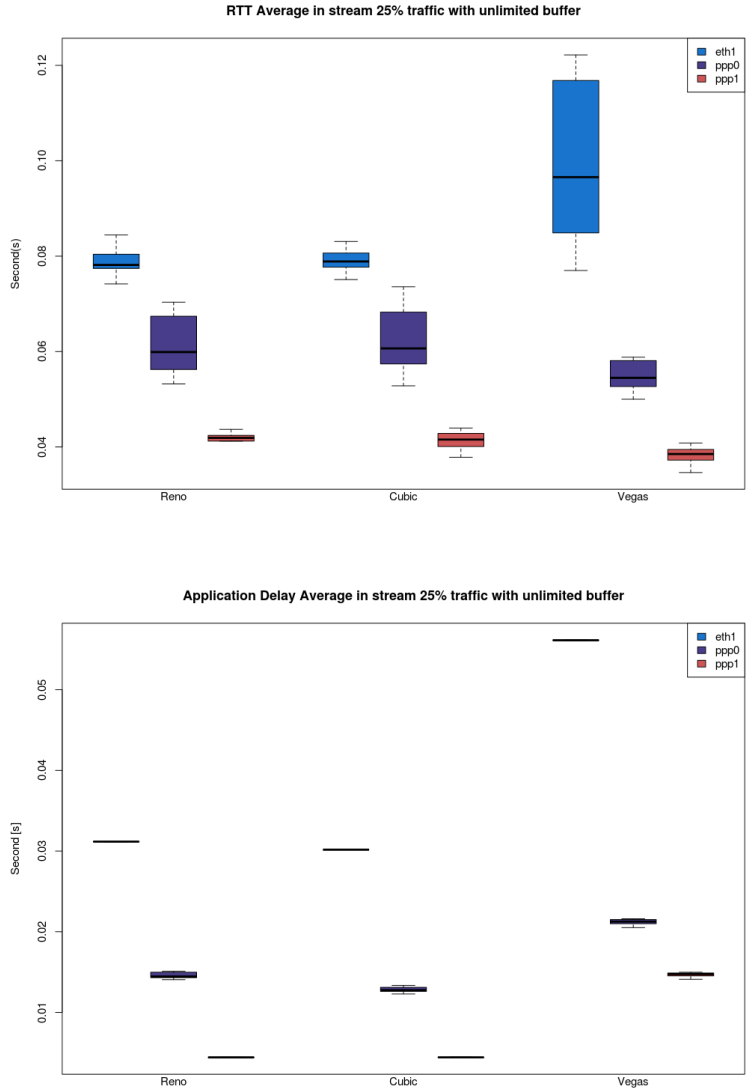


Figure 4.18: **Stream** traffic with 25% of bulk goodput rate unlimited buffer sizes

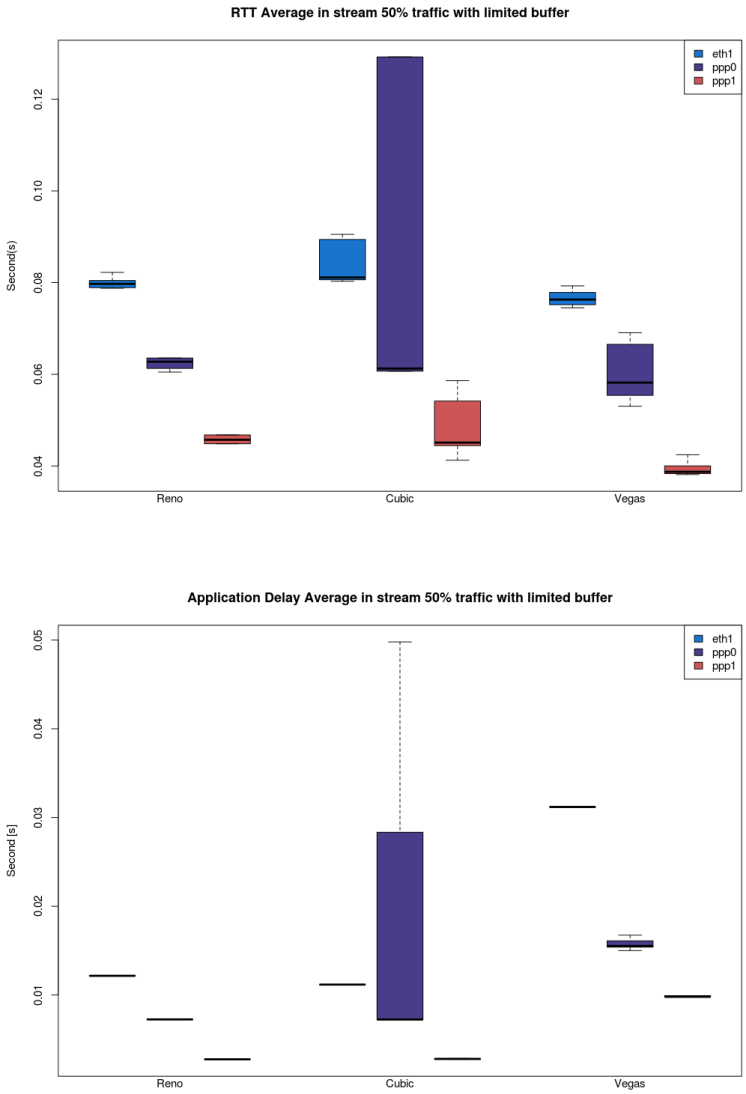


Figure 4.19: Stream traffic ith 50% of bulk goodput rate limited buffer sizes

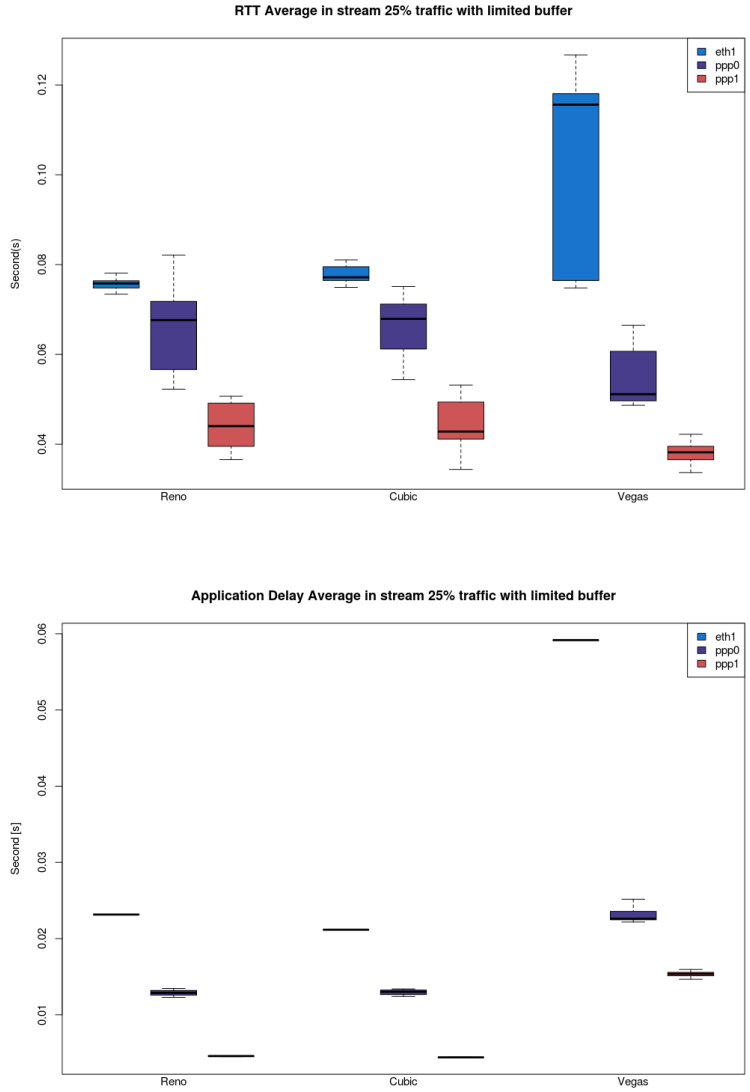


Figure 4.20: **Stream** traffic with 25% of bulk goodput rate limited buffer sizes

Chapter 5

Analysis

In this section the results from the measurement groups will be analysed in order to show the impact of the different TCP congestion control characteristics in QoS while they are running on different Mobile Broadband networks.

In this thesis, we are interested to see that by having the type of traffic that will be going to be used in TCP connection, how each congestion control algorithm performs over different Mobile Broadband networks and vice versa.

The following sections analyse the results based on the traffic types in which the behavior of the QoS's will be evaluated in different congestion control algorithms over different networks. The first section evaluates the *Bulk* traffic type following by next section in which the *Onoff* traffic will be evaluated. Finally the *Stream* traffic will be evaluated in the last section of this chapter.

5.1 Bulk traffic

In this section, the Bulk traffic will be analysed. As described earlier in Section 4.2.1, the sending rate is not limited. Hence, the sender utilizes the network by increasing the congestion window. Based on the type of the congestion control algorithm i.e, loss-based or delay-based the congestion window decreases when congestion occurs in the network. For loss-based algorithms i.e, Reno and Cubic, packet loss or timeout expiration is a sign of congestion in the network while Vegas as delay-based congestion control detects early congestion based on increasing RTT values.

As described earlier in Section 3.1.1, We conducted our measurements with two type of system settings i.e, buffer sizes. Unlimited (unbounded) buffers and Limited (bounded) buffers. The first was due to the fact that we wanted to test the protocol itself without any limitation from the system and the latter was according to simulate the behaviour of the TCP flavor in a real world. Therefore we chose Android buffer set sizes to simulate the

behavior of each TCP flavor in mobile devices platforms.

5.1.1 QoS in different congestion controls in same network

In this section we first try to analyze the behavior of the proposed congestion controls in each network and try to see the impact of congestion control in QoS by comparing different congestion control algorithm behavior in with each other in one network.

In the next section, we will try to investigate how each networks perform with one specific congestion control algorithm. Therefore we want to see how different providers treat our TCP connection which has same congestion control algorithm and system settings i.e, buffer sizes.

Unlimited buffer evaluation

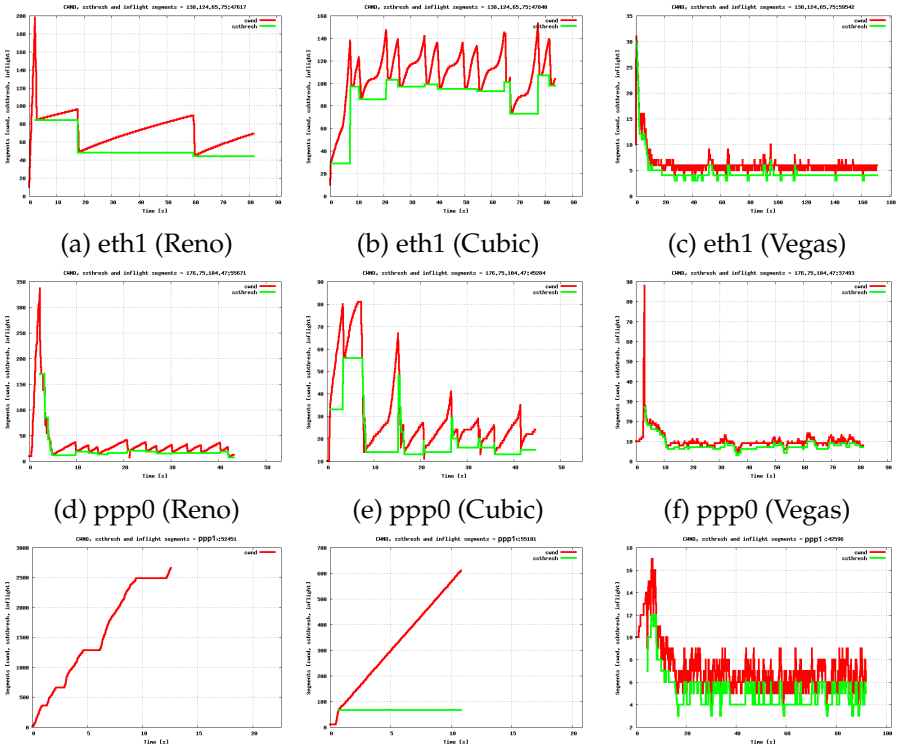
Figure 4.10 shows an overview of the results from measurement group $M_{b_{unlimited},t_{bulk}}$. The box plots show the distribution of the QoS values resulted from the repeated experiments in the measurement group.

In box plot, the samples are sorted. Then four equal sized groups are made from the ordered samples. That is, 25% of all samples are placed in each group. The lines dividing the groups are called *quartiles*, and the groups are referred to as *quartile groups*. The *median* (middle quartile) shows the mid point of the data and is shown by the line that divides the box into two parts. The median is usually close to the average. Half of the samples are greater than or equal to this value and half are less. The middle box represents the middle 50% of samples for the group. The range of samples from lower to upper quartile is referred to as the *inter-quartile* range. The middle 50% of samples fall within the inter-quartile range. Hence, 75% of the samples fall below the upper quartile and 25% of samples fall below the lower quartile. The upper and lower whiskers represent samples outside the middle 50%.

The *Average Goodput per second* box plots in Figure 4.10 show that ppp1 has approximately similar goodput rate in Reno and Cubic which is much higher than the other two networks in both Reno and Cubic congestion controls. However, in Reno, ppp1 has more variance in goodput value than it has in Cubic. Figure 5.1g shows the CWND evolution taken from one of the experiment in ppp1 network with Reno as congestion control with unlimited buffer size. As can be seen, since there is no packet loss in this network, Reno starts in *slow start* mode in which it increases the CWND exponentially and it won't go in *congestion avoidance* phase. Hence, the CWND value reaches to 2700 in the end of connection which means that in the last seconds of the connection, the sender sends roughly 2700 segments at each time. As a result, the RTT value which is shown in Figure 5.2g, increases relevantly to the growth of CWND and reaches to 4.5

seconds. This behavior of ppp1 network can be interpreted as being a large buffering on the path between known as Bufferbloat [1], in which all the sent segments are buffered without network getting congested while the RTT increases sharply. In contrast, the other two networks get quickly congested by increasement in the CWND value.

The CWND value of ppp1 network with Cubic congestion control (Figure 5.1h), Does not reach to the CWND value in Reno and at its peak it reaches to 600 segments in flight per second. This is due to the loss or changing to TCP-friendly CWND growth mode described in Algorithm 3 in which the CWND increases slowly than it does in Reno. By looking at the RTT value for this experiment in Figure 5.2h, it can be seen that since the CWND in Cubic is less than the CWND in Reno, therefore the queue of the routers are not overwhelmed. Hence, the RTT in Cubic is much lower than RTT in Reno (Figure 5.2g). In addition, since the Rate is calculated by dividing the CWND value to the RTT value, therefore the Goodput rate value of the Cubic is higher by roughly 100 KB/S.



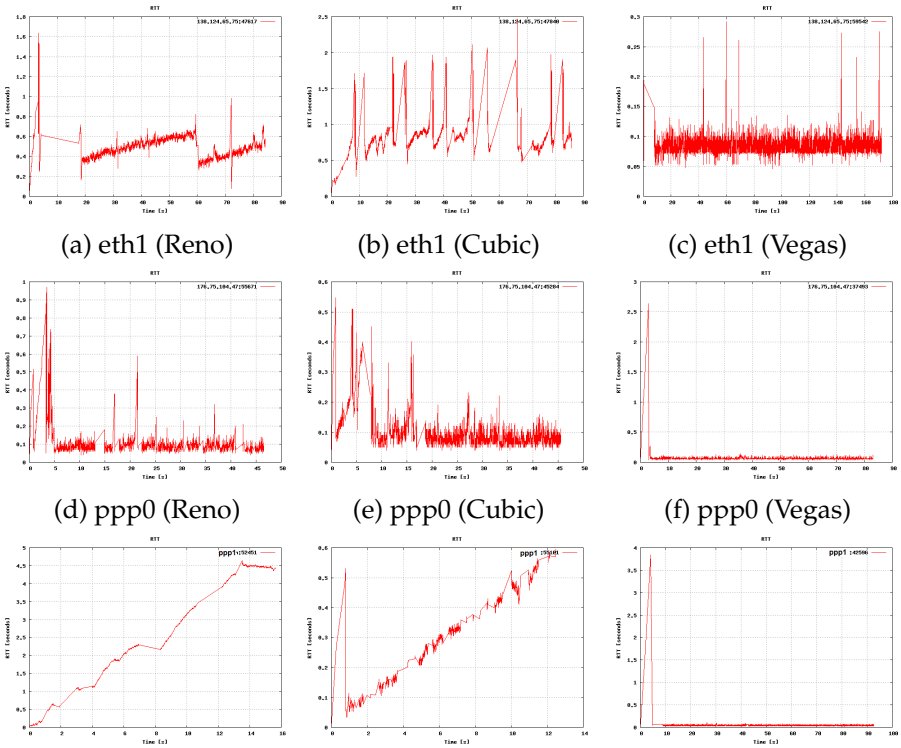
(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.1: CWND evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited bulk traffic

Table 5.1a shows that eth1 has similar goodput rate in both Reno and Cubic. We know that eth1 network , uses different technology and has lower capacity in case of bandwidth than the other two networks. Fig-



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.2: RTT evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited bulk traffic

	eth1	**ppp0**	**ppp1**
Reno	181.405	304.741	1137.788
Cubic	188.45	352.578	1273.132
Vegas	96.747	155.148	237.16

(a) Goodput [KB/s]

	eth1	**ppp0**	**ppp1**
Reno	0.675	0.449	1.732
Cubic	0.808	0.285	0.330
Vegas	0.092	0.076	0.050

(b) RTT [s]

	eth1	**ppp0**	**ppp1**
Reno	8.1	5.1	1.2
Cubic	9.6	4.2	1.0
Vegas	15	9.9	5.6

(c) One-way delay [ms]

	eth1	**ppp0**	**ppp1**
Reno	2	12	-
Cubic	14	26	1
Vegas	64.4	131.5	150.2

(d) Number of changes in ssthresh value

Table 5.1: The average values of Bulk traffic measurements with unlimited buffer size.

ure 5.1a shows the CWND starts in slow start phase, however due to the low capacity of the network, packet loss occurs when the rate reaches at 3 Mbit/s and Reno triggers the fast recovery. Hence, it starts in congestion avoidance phase and halves the ssthresh and CWND values as described

in Section 2.1.3. By going into congestion avoidance mode, the CWND growth will become slow and therefore it takes time for CWND to reach the value when the congestion happened although it has suffered from two more packet losses during the connection. Cubic acts more aggressively when tries to recover from congestion stage. As can be seen from Figure 5.1b, everytime the loss occurs the CWND in Cubic tries to reach the value at the time of last loss. Therefore the CWND value stays at value usually higher than it is in Reno. However, due to reaching the network capacity, it faces with more packet loss than Reno according to Table 5.1d. This shows that because of the limited link capacity, increase in CWND value makes the queue in the paths full and as a result the RTT value starts to increase. Hence, the rate is similar in both congestion controls.

The number of changes in ssthresh value could tell us that whether packet loss or timeout had occured in Reno and Cubic congestion control algorithms since they are loss-based. Table 5.1d shows that in all networks, the Cubic has more changes in ssthresh than Reno. However, as described above, it recovers faster and it is more aggressive than Reno. Therefore, the Goodput value in all three networks for Cubic is higher than Reno.

ppp0 acts similar to eth1 in Reno and Cubic, however due to the less delay in this network the rate in ppp0 with both Reno and Cubic is higher than eth1. Also, we know that our subscription in network ppp0 has downlink limitation rate to maximum 3 Mbps.

Vegas as a delay-based congestion control algorithm, tries to avoid delay in the network. Hence, it adjusts the CWND in order to maintain a small number of packets in the buffers of the routers in the path. Hence, the average delay in Vegas is much smaller. Vegas adjusts its CWND based on the queuing delay value which is the difference of the Base RTT and actual RTT of the sent packet. Base RTT is the minimum RTT measured in the path.

According to Table 5.1b, eth1 has higher delay than the two other networks. Therefore by increasing the rate in eth1, the queuing delay increases. Hence, Vegas tries to decrease the CWND in order to avoid the increase in the queuing delay in the network. As a result the RTT stays in a constant value which can be seen in Figure 5.2c. However the consequence of keeping the delay minimal is degradation in goodput per second, since the CWND value is relatively small comparing with two other congestion control algorithms. Figures 5.2f and 5.2i also show how Vegas maintains the delay at low values in the two other networks in order to avoid congestion in the network.

Table 5.1c shows the One-way delay values for our experiments in this group. Since we set our calculation of one-way delay on the differences in the received segments in goodput file, therefore the higher value in goodput per second value, means that the waiting time between receiving each

segment in one second interval is smaller. These can be seen by looking at Goodput average and Application delay box plots in Figure 4.10. The only exception is for eth1 with Cubic congestion control which has higher one-way delay than Reno while the goodput of eth1 in Cubic is higher than Reno .

From the box plots in Figure 4.10, it can be seen that although ppp1 has the higher goodput value than the two other networks in all congestion controls, it also has bigger variation in goodput, while ppp0 is the second network with highest variation in goodput value and eth1 is the network with the least variation in goodput. ppp0 also has the highest variance between two other networks in both Reno and Cubic congestion controls for RTT values.

The Equations (5.1) – (5.3), show the overview the considered QoS in each network with different congestion control.

$$RTT_{(Unlimited,Bulk)} \begin{cases} eth1_{Cubic} > eth1_{Reno} > eth1_{Vegas} \\ ppp0_{Reno} > ppp0_{Cubic} > ppp0_{Vegas} \\ ppp1_{Reno} > ppp1_{Cubic} > ppp1_{Vegas} \end{cases} \tag{5.1}$$

$$Goodput_{(Unlimited,Bulk)} \begin{cases} eth1_{Cubic} \geq eth1_{Reno} > eth1_{Vegas} \\ ppp0_{Cubic} > ppp0_{Reno} > ppp0_{Vegas} \\ ppp1_{Cubic} > ppp1_{Reno} > ppp1_{Vegas} \end{cases} \tag{5.2}$$

$$One-way\ delay_{(Unlimited,Bulk)} \begin{cases} eth1_{Vegas} > eth1_{Cubic} > eth1_{Reno} \\ ppp0_{Vegas} > ppp0_{Reno} > ppp0_{Cubic} \\ ppp1_{Vegas} > ppp1_{Reno} > ppp1_{Cubic} \end{cases} \tag{5.3}$$

Limited buffer evaluation

Figure 4.11, shows the results of bulk traffic measurements with the limited buffer set sizes as described in section 4.2.1.

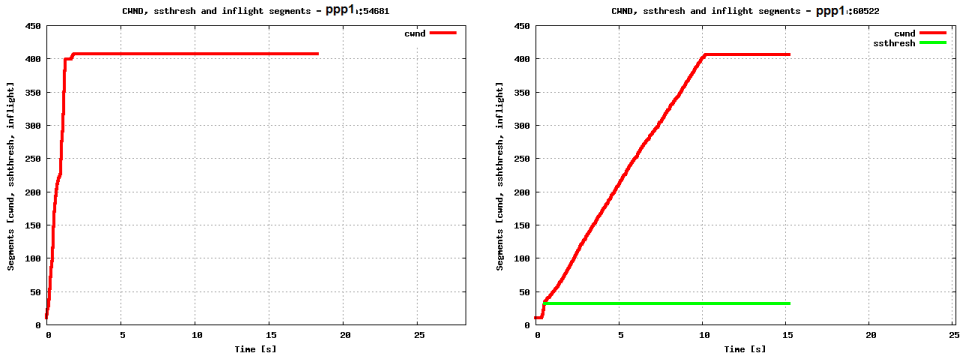
By limiting the buffer sizes on both server and node, the CWND growth will be limited to the amount of sender's sending buffer and receiver's receiving buffer size. This could clearly observed in case of ppp1 network - which has a higher bandwidth delay product - where with unlimited buffer sizes in Reno and Cubic (Figures 5.1g and 5.1h), the CWND keeps growing till the end of the connection. However, with limited buffer size the CWND does not pass the value of 400 segments in flight which is exactly correspond to the maximum buffer limit value that we have set for both sender and receiver. As an example, we calculate the rate in the second 10th of the selected experiment's CWND value in Bytes shown as Figure 5.1g and its corespondent RTT value from Figure 5.2g as following.

$$Rate_{10} = \frac{CWND_{10} \times MSS}{RTT_{10}}$$

$$Rate_{10} = \frac{400 \times 1500}{0.5}$$

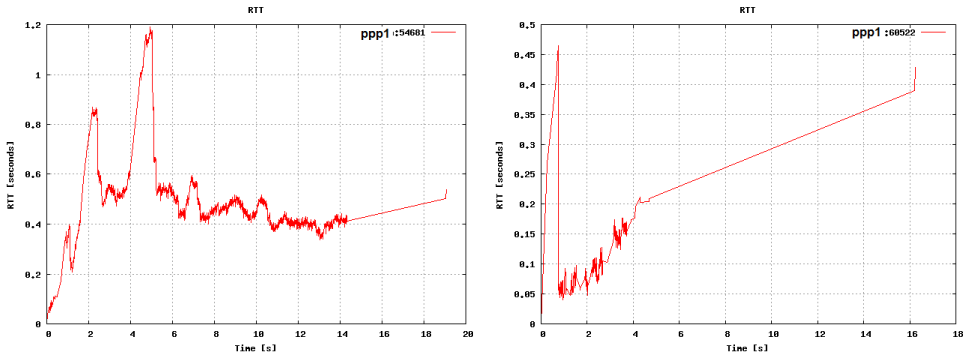
$Rate_{10} = 1200000$ Bytes per second $\equiv$ maximum limited buffer size in Table 4.1

The limitation in CWND value also results in smaller RTT value which can be seen by comparing Figures 5.3c and 5.2g.Table 5.2, shows the average QoS values resulted from bulk traffic with limited buffer size measurement group. By comparing the values to the values in Table 5.1 it could be possible to find the differences and the impact of the buffer size on the behavior of the congestion control algorithm in each network.



(a) CWND ppp1 (Reno)

(b) CWND ppp1 (Cubic)



(c) RTT ppp1 (Reno)

(d) RTT ppp1 (Cubic)

Figure 5.3: The CWND (first row) and RTT (second row) of ppp1 in Reno and Cubic congestion control with limited buffer size

According to the comparison between the unlimited buffer experiments and limited buffer sizes experiments, it can be seen that for Goodput, eth1 and ppp0 both have higher value with limited buffer size than they have in unlimited buffer size, while ppp1 has roughly similar goodput in Cubic and Reno with both limited and unlimited buffer sizes. In addition, Vegas has almost same goodput value in all networks with both unlimited and

limited buffer sizes.

The RTT values resulted in limited buffer size measurements, shows that the RTT values in all three networks with Reno and Cubic as the congestion controls are smaller than unlimited buffer size experiments while in Vegas, the RTT values are similar to the values resulted from unlimited buffer sizes.

From One-way delay values, it can be concluded that only eht1 in Reno and Cubic has got smaller one-way delay values while ppp0 and ppp1 both have the same one-way delay in Reno and Cubic with limited buffer as they have in unlimited buffer. Additionally, Vegas has almost the same One-way delay values in all networks with limited buffer as it runs with the unlimited buffer size.

In addition the number of ssthresh changes show that Reno has got more changes in both ppp0 and ppp1, while this value has been reduced in Cubic. The summary of the congestion controls impact on QoS values in each network are shown as Equations (5.4) – (5.6).

	eth1	**ppp0**	**ppp1**
Reno	252.571	359.850	1120.836
Cubic	270.437	361.136	1177.917
Vegas	91.079	134.556	220.186

(a) Goodput [KB/s]

	eth1	**ppp0**	**ppp1**
Reno	0.422	0.166	0.302
Cubic	0.534	0.149	0.198
Vegas	0.086	0.082	0.048

(b) RTT [s]

	eth1	**ppp0**	**ppp1**
Reno	6	4.6	1.1
Cubic	6.1	4.1	1.1
Vegas	15	10	6

(c) One-way delay [ms]

	eth1	**ppp0**	**ppp1**
Reno	3.8	14	-
Cubic	11	22	1
Vegas	71	156	152

(d) Number of changes in ssthresh value

Table 5.2: The average values of Bulk traffic measurements with limited buffer size.

$$RTT_{(Llimited,Bulk)} \begin{cases} eth1_{Cubic} > eth1_{Reno} > eth1_{Vegas} \\ ppp0_{Reno} \geq ppp0_{Cubic} > ppp0_{Vegas} \\ ppp1_{Reno} > ppp1_{Cubic} > ppp1_{Vegas} \end{cases} \tag{5.4}$$

$$Goodput_{(Limited,Bulk)} \begin{cases} eth1_{Cubic} > eth1_{Reno} > eth1_{Vegas} \\ ppp0_{Cubic} \geq ppp0_{Reno} > ppp0_{Vegas} \\ ppp1_{Cubic} > ppp1_{Reno} > ppp1_{Vegas} \end{cases} \tag{5.5}$$

$$One-way\ delay_{(Limited,Bulk)} \begin{cases} eth1_{Vegas} > eth1_{Cubic} = eth1_{Reno} \\ ppp0_{Vegas} > ppp0_{Reno} > ppp0_{Cubic} \\ ppp1_{Vegas} > ppp1_{Reno} = ppp1_{Cubic} \end{cases} \quad (5.6)$$

To summarize the behavior of the congestion controls in our three considered networks it can be said that the highest RTT value in eth1 which uses CDMA as its technology, is achieved by using Cubic while the two other networks which using UMTS technology have the highest RTT by using Reno as congestion control.

Since in Bulk traffic (i.e, transferring large files) , the main considered QoS could be Goodput per second value, therefore it can be concluded from the results that for bulk traffic with both limited and unlimited buffer size, the highest Goodput per second value could be achieved by using Cubic as congestion control algorithm in all three networks. Meaning that according to our study, regardless of the difference in the MBB technology, the Cubic congestion control has the highest Goddput per second in both UMTS and CDMA networks.

5.1.2 QoS with same congestion control over different networks

Until now, we have described the differences in QoS parameters of each network while they're running with different congestion control algorithms. However, we're mainly interested to see how a same TCP connection e.g, Reno, Cubic with the same characteristics e.g, buffer sizes perform over our different existed networks. This way of look gives us the ability to understand how different ISP's deal with every single congestion control algorithms.

From Figures 4.10 and 4.11 and Tables 5.1 and 5.2, the following facts could be concluded in congestion controls described in the following:

- **Reno:**

  ppp1 network has the highest Goodput per second value with the highest RTT and lowest one-way delay while it has roughly no loss. However it also has the bigger variance in Goodput and RTT and less variance in one-way delay. Limiting the buffer size only results in less goodput value comparing to unlimited buffer size.Hence, the RTT value is decrease and is less than eth1.

  ppp0 has the highest number of changes in ssthresh value which could be because of timeout or packet loss/reordering. However, in both limited and unlimited buffer sizes when the rate exponentially increases at the early moments of the connection, it follows with a packet loss which as a result the goodput rate decreases. This

could be somehow related to the fact that in this network the downlink bandwidth is limited to 3 Mbit /s. Therefore whenever the bandwidth-delay product is exceeding the limitation, a burst of consecutive losses occurs and as a result the rate will decreases. Figure 5.2d, shows that even with the low value of RTT, TCP Reno had several changes in ssthresh which is shown in Figure 5.1d. The limitation of the buffer size also results in having less RTT and one-way delay variance.

In eth1 network, the RTT is higher except in unlimited buffer size mode which ppp1 has the highest RTT. In addition, eth1 has the highest one-way delay among other networks in Reno. while the goodput is also low. After ppp0, this netwrk has the highest number of ssthresh changes from approximately 2 to 4 changes with unlimited and limited buffer respectively.

- **Cubic:**

  Cubic in eth1 has higher RTT value than two other networks, while this results to acquire less goodput among othe rtwo networks. It shows that whenever that the CWND reaches 140 segments, RTT reaches to roughly 2 seconds, which shows that the buffers in the path get filled and as the consequence a packet loss occurs. Although TCP Cubic quickly tries to reach to the CWND size at the time of congestion i.e, 140 but again with another loss its CWND decreases. The highest rate of delay i.e, RTT in this network makes it to have the smallest goodput per second comparing to other networks. In addition it has the highest one-way delay value.

  Same as in Reno, ppp0 starts the transmission by quickly reaching to high CWND value, but it quickly receives burst of packet losses/-timeout and as a result, the Goodput rate degrades. whenever the CWND tries to reach to higher value a loss happens. This behavior seems strange while the RTT does not reach a very high values. one possible reasons could be that ppp0 network is very lossy or the more possible answer is that the traffic shaper in the network tries to keep the downlink rate to the 3 Mbit/s. The one-way delay is less than eth1 and by limiting the buffer size the variances of the Goodput per second, RTT and One-way delay is also smaller.

  ppp1, has the highest Goodput value in Cubic congestion control among two other networks. However the RTT is very close to ppp0. The main reason of achieving higher goodput value is having only one change in ssthresh value which is due to loss or packet reordering. therefore, the bandwidth-delay product of this network is very high. In addition it has the lowest one-way delay which means that the transferred data were received back to back fastest than two other

providers.

- **Vegas:**

  Vegas minimizes the number of inflight packets in the queues on the path. Therefore as a result, the RTT of all networks are roughly similar and only differs by 20 ms. ppp1 has the highest goodput while eth1 has the least. ppp0 has higher variance in one-way delay. interestingly the one-way delay in Vegas for all networks is higher one-way delay values for Reno and Cubic for all three networks.

5.2 Onoff traffic

As described earlier in Sections 3.1.1 and 4.2.2, The Onoff traffic generated by NetPerfMeter tool in order to simulate the short flow traffics. In which for a limited period of time the sender sends traffic with unlimited sending rate. meaning that the it sends as much segments as possible in the specified short period of time and then stays idle for another specified amount of time. The way we conducted our measurements in this type of traffic, in the duration of 60 seconds connection, the sender starts sending segments for 1 second and then stays idle for 5 seconds, Therefore in the duration of each measurement, there are normally 10 short flow traffics generated and sent by the server. Although our specified sending duration is just 1 second, but due to the system overheads, each sending duration roughly takes 1-3 for the receiver to receieve all the segments.

Same as measurements in long flow traffic (i.e, bulk), we measured the short flow traffic with both unlimited and limited buffer sizes.in following subsections, we first analyze the results from running the measurements with Unlimited buffer sizes and next will be the results from limited buffer sizes.

5.2.1 QoS in different congestion controls in same network

Unlimited buffer evaluation

Figure 4.13 shows the box plots resulted from the Onoff traffic with unlimited buffer size measurement group.

As can be seen from Figure 5.4a, TCP Reno starts in slow start phase from second 0 which is the start of the connection and CWND quickly reaches to 140 segments, after that the server stays idle and doesn't send any traffic (the steady line from second 1st to 6th in CWND plot). Upon starting the second period it can be seen that the CWND starts from 140 segments and continues in the slow start phase since there was no loss and no ssthresh were set. From RTT plot in Figure 5.5a, it can be seen that RTT increases meaning that the network buffers are filling up and hence the packet loss occurs and CWND got halved. from this moment TCP Reno,

goes to the congestion avoidance phase which increases the CWND by one in each RTT. Hence, for the rest of the onoff periods, the increase of CWND is slower.

However, in Cubic (Figure 5.4b), although more packet loss occurs, but it tries to quickly recover and increase the CWND size. Hence, the RTT (Figure 5.5b) increases too due to the less bandwidth and capacity of this network. As a result the Goodput per second rate of Cubic in eth1 is less than Reno.

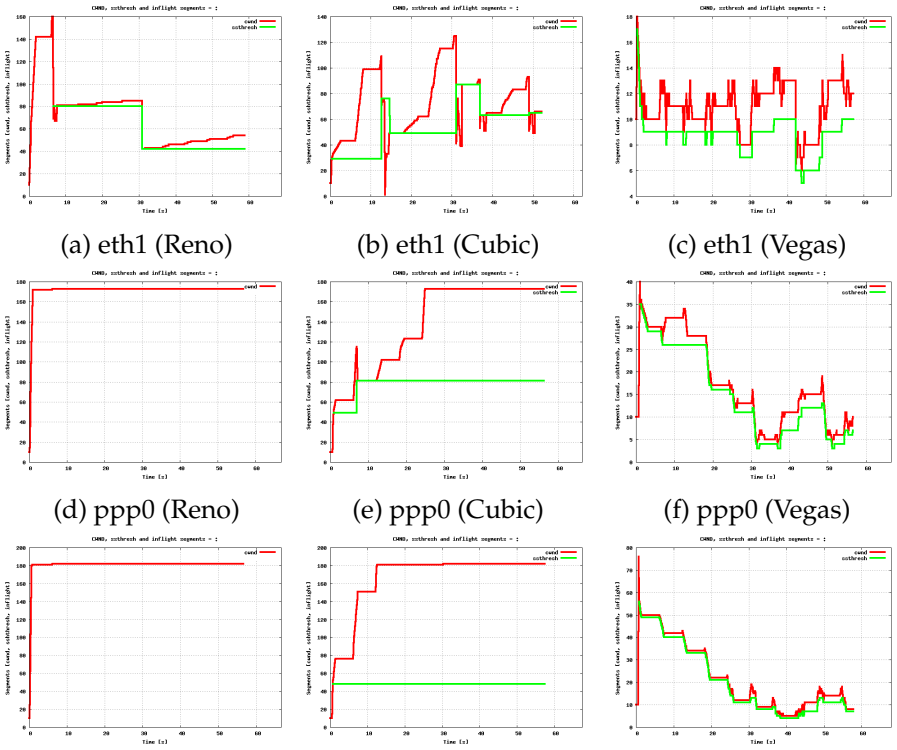
TCP vegas increases the CWND if the difference between the base RTT and the measured RTT which is called queuing delay is too small. In eth1 (5.5c) , TCP vegas starts increasing the CWND upon first sending interval and by increasement in RTT, it quickly decreases the CWND and updates its Base RTT value. during the 5 seconds of non sending traffic, the queues in the path were emptied and by starting the next sending interval, Vegas increases the CWND value since the queuing delay is small. In eth1 network, since the capcaity of the network is not high comparing to two other networks, vegas gains more goodput than the Reno and Cubic by not filling the buffers of the routers in the path. while in Cubic and Reno there are packet losses because of this phenomenon and as a result they suffer higher RTT value than Vegas.

In ppp0 and ppp1, due to the higher link capacity, the CWND value reaches to 170 and 180 respectively for both Reno and Cubic. However, in Cubic an early increasement in RTT value which is due to the sharp increase in CWND value, results in loss. Therefore the CWND reaches to its maximum size after 2nd or 3rd sending period. The reason for having steady CWND value once it reaches the maximum value is the fact that since the server only transmits the data for 1 second, therefore all the data that server can transmit for that 1 second could be send by that CWND value, therefore during that 1 second if no loss happens, the congestion control won't increase the CWND anymore since it has already sent all the possible segments in that 1 second.

From Figure 4.13 and Table 5.3, it can be seen that in eth1 Cubic has higher RTT than Reno while both have big variances. However, the delay time for ppp0 and ppp1 in both Reno and Cubic are roughly same with the assumption that ppp1 has more variance than ppp0. For Vegas, all three networks have roughly same delay time.

For Goodput per second, eth1 has the highest rate with Vegas as congestion control algorithm while Cubic achieved lowest rate. However, in both ppp0 and ppp1 networks, the highest Goodput rate is achieved by Reno and Vegas has the lowest rate. Also ppp1 has the highest Goodput rate variance in all congestion controls.

From Application delay (One-way delay) plot, it can be seen that eth1 has the highest one-way delay of roughly 10 ms in all three congestion control algorithms comparing the two other networks. While in ppp0 and ppp1 , the highest one-way delay is when they run with Vegas as their con-



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.4: CWND evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited onoff traffic

	eth1	ppp0	ppp1
Reno	68.642	105.063	139.607
Cubic	61.592	97.291	113.925
Vegas	71.133	89.270	102.293

(a) Goodput [KB/s]

	eth1	ppp0	ppp1
Reno	0.467	0.335	0.213
Cubic	0.651	0.397	0.254
Vegas	0.108	0.100	0.074

(b) RTT [s]

	eth1	ppp0	ppp1
Reno	11	3.3	1.8
Cubic	10	3.7	3.2
Vegas	12	5.4	3.8

(c) One-way delay [ms]

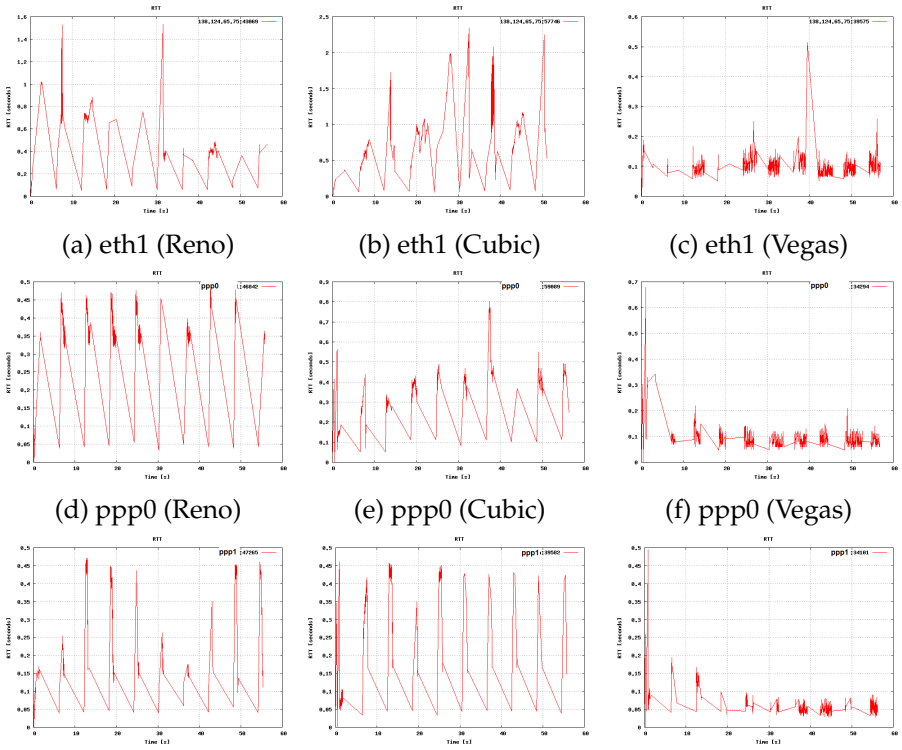
	eth1	ppp0	ppp1
Reno	2	-	-
Cubic	6	1.5	1
Vegas	28.4	48.3	57.5

(d) Number of changes in ssthresh value

Table 5.3: The average values of onoff traffic measurements with unlimited buffer size.

gestion control algorithm.

Additionally, Table 5.3d, shows that only eth1 has changes in ssthresh value by using Reno and has more changes in Cubic than ppp0 and ppp1.



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.5: rtt evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited onoff traffic

The summary of the congestion controls impact on QoS values in each network are shown as Equations (5.7) – (5.9).

$$RTT_{(Unlimited,On)} \begin{cases} eth1_{Cubic} > eth1_{Reno} > eth1_{Vegas} \\ ppp0_{Cubic} > ppp0_{Reno} > ppp0_{Vegas} \\ ppp1_{Cubic} > ppp1_{Reno} > ppp1_{Vegas} \end{cases} \tag{5.7}$$

$$Goodput_{(Unlimited,Onoff)} \begin{cases} eth1_{Vegas} > eth1_{Reno} > eth1_{Cubic} \\ ppp0_{Reno} > ppp0_{Cubic} > ppp0_{Vegas} \\ ppp1_{Reno} > ppp0_{Cubic} > ppp0_{Vegas} \end{cases} \tag{5.8}$$

$$One-way\ delay_{(Unlimited,Onoff)} \begin{cases} eth1_{Vegas} > eth1_{Cubic} > eth1_{Reno} \\ ppp0_{Vegas} > ppp0_{Cubic} > ppp0_{Reno} \\ ppp1_{Vegas} > ppp1_{Cubic} > ppp1_{Reno} \end{cases} \tag{5.9}$$

Limited buffer evaluation

By limiting the buffer sizes values, it can be seen that although the overall patterns has not changed, the variances of the experiment results are bigger. Although that the goodput results of each network in different congestion controls are roughly similar, However, it can be seen from Figure 4.14 and Table 5.4 that the Goodput rate in eth1 with Reno is slightly more than Vegas.

	eth1	**ppp0**	**ppp1**
Reno	85.385	102.762	137.953
Cubic	80.871	86.784	116.577
Vegas	82.947	88.594	100.689

(a) Goodput [KB/s]

	eth1	**ppp0**	**ppp1**
Reno	0.300	0.456	0.205
Cubic	0.420	0.465	0.255
Vegas	0.089	0.120	0.075

(b) RTT [s]

	eth1	**ppp0**	**ppp1**
Reno	6	3.2	1.8
Cubic	6	3.7	2.7
Vegas	6.4	5.5	3.8

(c) One-way delay [ms]

	eth1	**ppp0**	**ppp1**
Reno	2	-	-
Cubic	7	1.3	1
Vegas	31.25	54.5	52.5

(d) Number of changes in ssthresh value

Table 5.4: The average values of onoff traffic measurements with limited buffer size.

5.2.2 QoS with same congestion control over different networks

According to the results from both limited and unlimited buffer sizes we can summarize the impact of each congestion control on QoS parameters while they run on different MBB networks.

In Onoff traffic the delay is important QoS parameter. Therefore in this section we will evaluate RTT and One-way delay to see how different providers impact on these parameters in each congestion control algorithm.

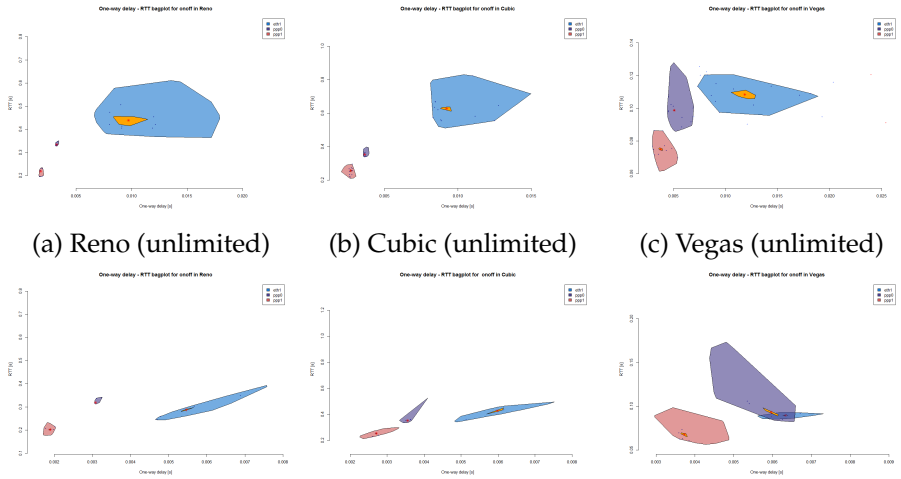
Figure 5.6, evaluates the impacts of each TCP congestion control in all three different MBB providers. The Bag plots [63] which were shown here, consist of two polygons, the inner polygon which is called bag has the median along with 50% of all data points and the outer polygon which is called loop contains observations between the bag and the fence. The fence is the inflation of the bag by a factor 3. Any observation outside the fence is outlier. In Figure 5.6, only the bag and loop were shown. The X axis represents the One-way delay values i seconds while the Y axis represents RTT values in seconds.

From Figures 5.6a and 5.6d it can be seen that in Reno, ppp0 and ppp1 are roughly similar where ppp0 has the lowest RTT delay and One-way

delay meaning that the application gets better performance by having more bandwidth which is due to the less RTT delay, and also the small one-way delay time which can be caused by packet reordering or recovering from loss. However, it can be seen that eth1 has highest delay both in RTT and One-way delay, therefore the application which is in our case could be the web browser, would suffer more delay comparing with other two networks. Also it shows that limiting the buffer size sets increase the variance of the One-way delay variance while it decreases the RTT variance in eth1.

Cubic congestion control behaves roughly similar to Reno in all three networks in both limited and unlimited buffer set sizes. however, comparing to Reno, ppp0 and ppp1 have slightly more variance in one-eay delay time in Cubic (Figures 5.6b and 5.6e).

Figures 5.6c and 5.6f show that TCP Vegas achieved less delay and less One-way delay while it runs over ppp1. However, eth1 and ppp0 have gained roughly same amount of RTT delay and eth1 has higher One-way delay than ppp0. In limited buffer sizes the RTT variance is higher for all networks and as it shows, although ppp0 has more variance in RTT than eth1 but their median are equal.



(a) Reno (unlimited) (b) Cubic (unlimited) (c) Vegas (unlimited)

(d) Reno (limited) (e) Cubic (limited) (f) Vegas (limited)

Figure 5.6: Bagplots of RTT-One way delay of networks in each congestion control

5.3 Stream traffic

As described earlier in Sections 3.1.1 and 4.2.3 and same as Onoff traffic, the NetPerfMeter were used in order to generate this type of traffic. The aim of generating this traffic was to simulate the behavior of the congestion controls in the networks while they're dealing with application such as media

streaming or VoIP in which the rate is limited by the application. Therefore we tried to use the Goodput rate of the bulk measurements in which the link were saturated and hence, generated the stream traffic by limiting the rate via NetPerfMeter to 50% and 25% of their corespondent experiments in bulk traffic.

Since in this traffic we have specified the Goodput rate manually therefore the Goodput rate is constant. Hence, we will not evaluate the Goodput rate for Stream traffic. Thus, we are mainly interested for RTT and One-Way delay as QoS in Stream traffic.

According to the different Goodput rates resulted from unlimited and limited bulk traffic from Tables 4.2 and 4.3. The Goddput rates which were considered for stream with 25% and 50% rate are differ in limited and unlimited buffer sizes. Therefore we first analyze and compare the stream 50% and 25% with unlimited buffer size and the two other groups with limited buffer size together.

By comparing the Figures 4.17 and 4.18 and looking at Tables 5.5 and 5.6, it can be seen that eth1 has the highest One-way delay in all congestion control algorithms. Also from Figures 5.8 and 5.10 it can be depicted that although the delay values are very close to each other, but eth1 has some relatively high spikes in RTT figures in all congestion controls.

ppp1 has the lowest One-way delay in all congestion controls with similar value in Reno and Cubic. In the other point of view, it shows that TCP Vegas has the highest One-way delay among all congestion controls while running on all networks.

	eth1	**ppp0**	**ppp1**
Reno	87.611	136.736	446.331
Cubic	93.266	171.812	453.092
Vegas	48.511	73.815	115.001

(a) Goodput [KB/s]

	eth1	**ppp0**	**ppp1**
Reno	0.077	0.079	0.049
Cubic	0.094	0.152	0.046
Vegas	0.076	0.062	0.041

(b) RTT [s]

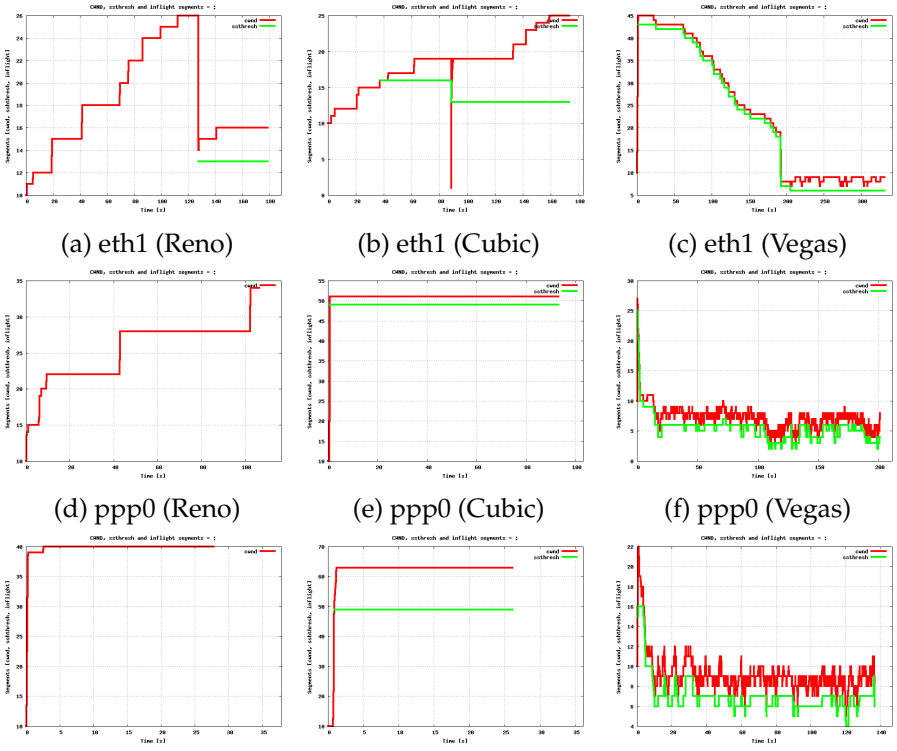
	eth1	**ppp0**	**ppp1**
Reno	16	9	2.7
Cubic	15	7.3	2.7
Vegas	29	15	9.6

(c) One-way delay [ms]

	eth1	**ppp0**	**ppp1**
Reno	2	0.5	-
Cubic	3	1.8	1
Vegas	29	114	92.5

(d) Number of changes in ssthresh value

Table 5.5: The average values of stream 50% traffic measurements with unlimited buffer size.



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.7: CWND evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited stream 50 traffic

	eth1	ppp0	ppp1
Reno	45.374	71.758	274.191
Cubic	46.880	87.569	274.554
Vegas	25.172	38.809	58.528

(a) Goodput [KB/s]

	eth1	ppp0	ppp1
Reno	0.079	0.073	0.041
Cubic	0.079	0.062	0.041
Vegas	0.100	0.057	0.038

(b) RTT [s]

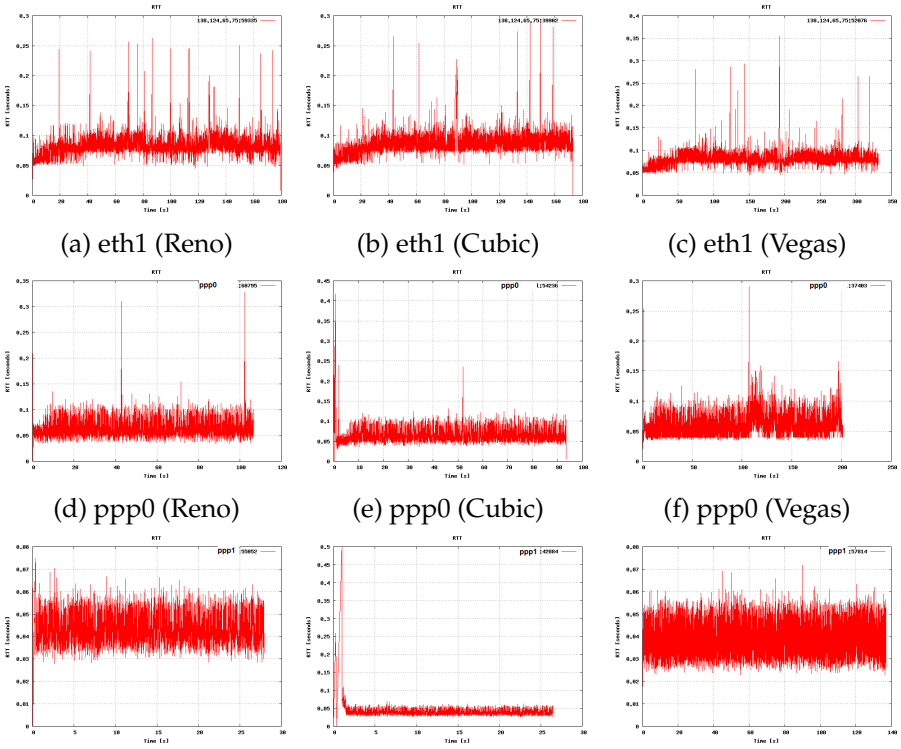
	eth1	ppp0	ppp1
Reno	31	15	4.4
Cubic	30	12	4.4
Vegas	56	21	14

(c) One-way delay [ms]

	eth1	ppp0	ppp1
Reno	1.5	1.3	-
Cubic	1.9	1	1
Vegas	93.7	28.4	5.9

(d) Number of changes in ssthresh value

Table 5.6: The average values of stream 25% traffic measurements with unlimited buffer size.



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.8: RTT evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited stream 50 traffic

	eth1	**ppp0**	**ppp1**
Reno	116.145	170.609	440.455
Cubic	126.529	172.681	405.086
Vegas	45.359	66.840	106.085

(a) Goodput [KB/s]

	eth1	**ppp0**	**ppp1**
Reno	0.081	0.091	0.063
Cubic	0.088	0.069	0.561
Vegas	0.076	0.060	0.040

(b) RTT [s]

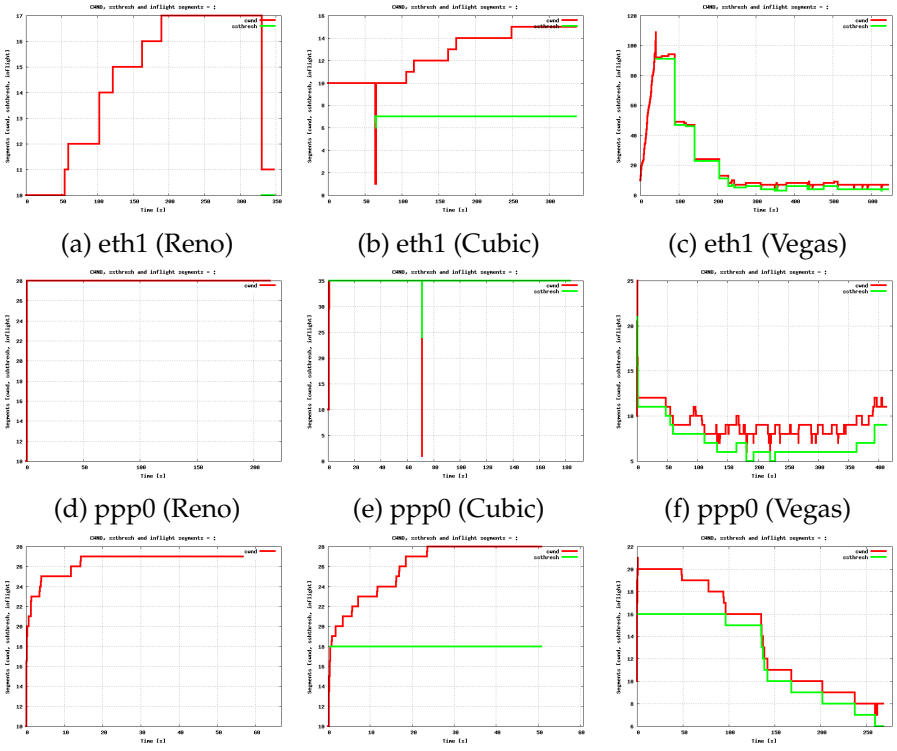
	eth1	**ppp0**	**ppp1**
Reno	12	7	2
Cubic	11	7.2	59
Vegas	31	15	10

(c) One-way delay [ms]

	eth1	**ppp0**	**ppp1**
Reno	1	1	-
Cubic	3	1.6	1
Vegas	18.8	121.8	74.25

(d) Number of changes in ssthresh value

Table 5.7: The average values of stream 50% traffic measurements with limited buffer size.



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.9: CWND evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited stream 25 traffic

	eth1	ppp0	ppp1
Reno	61.076	87.563	264.693
Cubic	66.836	85.510	273.794
Vegas	23.895	33.118	54.037

(a) Goodput [KB/s]

	eth1	ppp0	ppp1
Reno	0.076	0.065	0.044
Cubic	0.077	0.084	0.046
Vegas	0.101	0.065	0.044

(b) RTT [s]

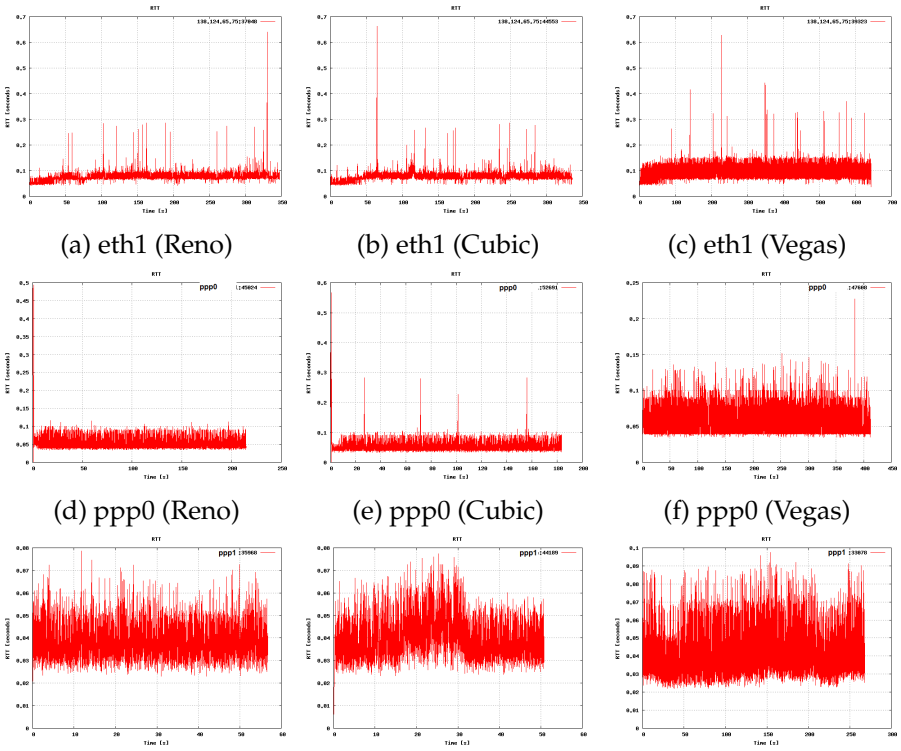
	eth1	ppp0	ppp1
Reno	23	12	4.5
Cubic	21	13	4.4
Vegas	59	23	15

(c) One-way delay [ms]

	eth1	ppp0	ppp1
Reno	1	-	-
Cubic	1.6	2.7	1
Vegas	134	18	12

(d) Number of changes in ssthresh value

Table 5.8: The average values of stream 25% traffic measurements with limited buffer size.



(a) eth1 (Reno) (b) eth1 (Cubic) (c) eth1 (Vegas)

(d) ppp0 (Reno) (e) ppp0 (Cubic) (f) ppp0 (Vegas)

(g) ppp1 (Reno) (h) ppp1 (Cubic) (i) ppp1 (Vegas)

Figure 5.10: RTT evolution of eth1, ppp0 and ppp1 in selected experiment with Unlimited stream 25 traffic

Chapter 6

Discussion and future works

The aim of this thesis was to observe the impact of different proposed congestion control algorithms on the Quality of Services over different Mobile Broadband networks from different types of traffic point of view. The design of the project which consisted of the design of experiments and measurement tools along with the results of measurement groups containing different experiments and analysis of the results have been presented in the previous chapters. In this chapter the measurement tool implementation, practical results and analysis will be discussed. In addition any probable weakness of the measurement methodology, possible adjustments and possible future works will be discussed as well.

6.1 Implementation overview

In this section we will discuss about the measurement tools and the methodologies that have been used in order to answer our problem statements.

6.1.1 Metrics

A set of different variables were considered in the measurements in order to observe the differences. Hence, we chose three different congestion control algorithms which use different strategies for avoiding congestion in the network. To further improve our comparisons we decided to use both *loss-based* and *delay-based* congestion control algorithms. In loss-based algorithms, packet loss in the network is interpreted as a sign of congestion in the network, while in delay-based algorithms an increase in the RTT of the sent packets is interpreted as sign of congestion.

TCP Reno and *TCP Cubic* were used as the two loss-based congestion control algorithms. Reno is the standardized congestion control algorithm by IETF and is used as a baseline in researches for comparisons. Cubic is the default congestion control algorithm for Linux as of kernel version 2.6.26 . Both algorithms are the most widely used algorithms in today's Internet according to Table 2.1. In addition, TCP Vegas were used as the

delay-based congestion control algorithm. All three algorithms have their own characteristics and methodology to how to deal with congestion in the network.

Different types of traffics were also considered in our experiments in order to observe the behavior of the stated congestion control algorithms while they faced with various traffic types. Hence, we proposed three different traffic types as *Bulk*, *Onoff* and *Stream*. In bulk traffic we tried to simulate the transferring of the large amount of data (i.e, 16 MBytes) in order to simulate the cases where a file is downloading using FTP or SCP. This could also referred as long flow traffic in which the sender tries to increase the sending rate based on the advertised receiving window from receiver and its own congestion window until the congestion in the network detected which based on the type of the algorithm the sending window size and rate will be adjusted. The Onoff traffic could be referred as short flow traffic in which the sender sends as much data as possible according to its sending window for limited amount of specified time and then it stays idle for another specific time. The goal for using this traffic was to somehow simulate the user experience while surfing the web pages, in which the web page is downloaded from the server (sending period i.e On) and then user normally tries to read the page for some amount of time (idle period i.e, off). We set the On period as 1 second and Off period as 5. Normally in this type of traffic the link will not utilized since the duration of On period is too small. Finally we conducted another type of traffic as Stream traffic which could be referred as Application limited traffic. Some traffics such as media streaming traffic or VoIP traffics use a limited rate of sending traffic over the network. Therefore we used the method that we used in bulk traffic but this time with limiting the rate of the sending data from the server by 50% and 25% of the maximum rate achieved in bulk traffic type to simulate the Stream traffic.

In order to see the protocol behaviour with and without system settings limitation, we considered two types of system setting in which the socket's write and receive memory set sizes are different. for unlimited buffer size, we set the maximum send and receive buffer size equal to the amount of data that were used in bulk traffic (i.e, 16 MB) which comparing to the default value of the Linux (i.e, 0.5) could be referred as Unlimited. In addition for limited buffer size we used the Android's buffer sizes value for HSPA+ connection. Table 4.1 shows the buffer sizes that were used in this thesis.

Three Mobile Broadband providers of Norway with different 3G technologies were used as our networks. One with 3G-CDMA2000 1xEV-DO Rev.A technology mentioned as *eth1* in this report with theoretical data rate of 9.3 Mbit/s in downlink and 3.1 Mbit/s in uplink. The other two networks were both using 3G-UMTS with HSPA+ mentioned as *ppp0* and *ppp1* with theoretical data rate of up to 21.6 Mbit/s in downlink and 5.8 Mbit/s in uplink. However, the *ppp0* data subscription is limited to 3Mbit/s in downlink as stated in our subscription. Additionally we only conducted

our studies based on downlink and not uplink.

The QoS metrics which we were interested in this study were *Goodput*, *RTT* and *One-way delay (i.e, application delay)*.

6.1.2 Experiments

Having the metrics and variables , we designed our experiments by grouping them based on the type of traffics with limited and unlimited buffer sizes as sub group of each traffic. In each group, one congestion control algorithm were set at the server which is the sender in our study and the measurement will executed, this pattern repeated until the measurements are done with one congestion control for all three networks. After that the next congestion control were used in the networks and so on. Therefore each measurement group consists of 9 experiments.

6.1.3 Measurement tools

In order to observe the behavior of each connection and extract the connection parameters, a set of tools were created which consisted of two scripts. One were executed at client and the other one at the server. The server script set the congestion control and the buffer size and captures the packets while listening to the incoming socket connection from the client. Also the client script capture the transferred packets and initiates the connection to server. Based on the traffic type, the server generates the traffic to transmit to the client. The outcome of the measurement scripts were 4 files in which two of them are trace files both from the packet capturing of server and client and one Goodput file which client script creates based on the amount of data it receives at each time through the socket which could be interpreted as Goodput values since it shows the data that application has received without any protocol overhead and etc.

6.1.4 Collected data

In total, 72 individual experiments have been proposed in this thesis. However, each experiment were repeated several times in order to take the distribution of the QoS values. The mean of QoS values from each repeated experiment were recorded in files along with the experiment name. Hence, all 72 experiments represent a distribution of sampling means for Goodput, RTT and One-way delay from repeated experiments. These files were used for plotting the results of each measurement group.

6.2 Results and Analysis overview

The results of the experiments were evaluated in two different ways. In one way, we tried to compare the impact of different congestion control algorithms on QoS over each individual network. In the other way, we evaluated the impact of each TCP connection with individual congestion

control algorithm while running over different networks. In the first way of analysis, by looking at the boxplots and tables based on an individual network in different congestion control algorithms some results and behaviors arised our attention as following.

- In almost all traffic types in all networks, Vegas has the lowest Goodput rate comparing to other congestion control algorithms. However, in eth1 (3G-CDMA2000 network) with Onoff traffic with both limited and unlimited buffer sizes, Vegas has the higher Goodput than Reno and Cubic.
- Since Vegas is a delay-based congestion control algorithm and as the nature of its mechanism, it tries to minimize the RTT (i.e, delay) in the network. Interestingly it has the highest One-way delay (i.e, application delay) value in all networks among other congestion controls for almost all kind of traffics.

6.3 Future Works

In this Section, the the future works that could not be done in this project due to the time limit and the potential possible projects based on the results of this thesis are explained.

6.3.1 Finding traffic manipulation by providers

Our study showed that the loss based congestion controls such as Reno, face with sharp decrease in CWND size whenever that a packet loss ocuurs in the network. However, it is possible that some providers implement some Active Queue Management (AQM) system which could randomly drop some packets in the routers which are exist in the path. As a result, the senders which are using Reno as congestion control will mistakenly think that there is a congestion in the network and as a result they decrease the CWND and the throughput degradation is the consequence.
However, lots of measurements and analysis is needed in order to find and prove this scenario, which was out of this thesis scope.

6.3.2 Bufferbloats

Our results showed that one of our used networks (ppp1) is equipped with large amount of buffer. However, more experiments and measurement were needed in order to be able to fully understand the behaviour of each congestion control in this large buffers known as bufferbloats.

6.3.3 Multipath congestion control

The results of this project could be used for improvements in multi-path transport, e,g. for scheduling of data onto paths as well as for multi-path congestion control particularly for CMT-SCTP [36] and MPTCP [30] protocols.

6.4 Potential weaknesses and improvement adjustments

In this section the potential weaknesses of the methodology which have been proposed in this thesis along with possible modifications for better accuracy will be described.

6.4.1 Repetition of experiments

As we described earlier, the measurement procedure in this thesis consisted of 72 individual experiments with different characteristics. However, in order to have reliable sampling distribution of these experiments, each of them should get measured for reasonable number of times (more than 30 is desirable). In addition there should be considerable amount of time between each experiment which were going to take place on each network in order to have the results of the experiments independent than each other and avoid the caching or buffering the route and other metrics in the underlying routers in the path. Therefore, each measurement group consisting of 9 experiments were executed only two times per working days. One in the beginning of working hours and the other in the evenings at the end of working hours.

Additionally, the Mobile Broadband subscriptions have limited and expensive monthly data plan. which means that there is a quota for the amount of sent/received data in each month. Therefore, although having more than two measurements per day is logically none sense due to the same results that it will provide, it would be even costly and expensive because of the quota and monthly subscription.

In order to tackle this problem and having as much repeated experiments as possible, a longer period of measurements is required. In the other words, if we run the measurements during 4 to 5 months, then the results would be accurate enough to be heavily relied on.

6.4.2 One-way delay measurement

As we described earlier on page 4.1.2, the correct way of measuring the One-way delay is by using the timestamp option in TCP header enabled by the sender for each packet in which the server prints the epoch time of the moment that the segment is being transferred. Thus, the receiver can find the exact amount of time by comparing the captured packets times with the timestamp printed in the header of the packets. However, both sender and receiver in this method should be accurately synced by using NTP protocol for instance. But since the One-way delay value is usually few milliseconds and the NTP accuracy is not better than 10 or 20 ms for WAN networks and is closely dependent on the machine's hardware and CPU clock specification, some other workaround were needed by using GPS devices which is

described in [4].

However, due to the limitation and lack of time the method proposed in [4] could no get implemented in our project. Hence we decided to measure the differences of each two timestamps in the Goodput file. this difference shows the amount of time that took for the application to receive the next segment from transport layer. This could also gives us the information about the delay which were cause by loss or reordering of the packets in transport layer.

One-way delay values from NetPerfMeter

As we described in previous section, we used the difference of each two timestamps from the Goodput file which was the output file of the client measurement script. However, we used NetPerfMeter for generating customized traffics i.e, Onoff and Stream. Since we couldn't have access to the socket which being used by NetPerfMeter in order to extract the Goodput values, therefore, we used the vector files which NetPerfMeter generated. As shown in 4.1.2, this file summarizes all the amount of data that has been sent from the server to the client in each 1 second interval. Hence if we were going to use the differences between these timestamp values, all the One way delay results would have a constant value equal to 1.

The only workaround that we could implement in the short amount of time that we had, was to use the client trace files and filter them in a way to show the timestamp of each received segment in epoch format. Then by taking the differences between these timestamps we measured the One-way delay values same as Bulk traffic.

Chapter 7

Conclusion

In this thesis, we studied the TCP Reno, TCP Cubic and TCP Vegas congestion control algorithms and their impact on QoS characteristics over operational 3G-UMTS and 3G-CDMA2000 1xEV-DO Rev.A networks in Oslo, Norway. We showed that the same TCP connection (i.e, with one congestion control, system settings and etc.) have different impacts on the Quality of Services parameters in each network even if the underlying technology of the networks are same.
Our results could be categorized based on the traffic types as following:

- Bulk traffic:

  Our results show that, TCP Cubic achieves the highest Goodput in both 3G-UMTS and 3G-CDMA2000 networks regardless of buffer sizes. Also TCP Vegas has the least Goodput in all networks. Our results also show that the networks with higher loss will have less delay in Reno, hence will have better Goodput comparing to the networks which have fewer loss and higher delay. However, in loss based algorithms if a network has no loss and high delay, then the Goodput also will be higher. It can be depicted that the networks with higher delay and less Goodput have higher application delay too.

- Onoff traffic:

  In onoff traffic, the networks with higher delay and more loss acheive the least Goodput and highest One-way delay in Reno and Cubic However, in Onoff traffic the 3G-CDMA 2000 network acquires the highest Goodput while uses Vegas as its congestion control while both 3G-UMTS networks have highest Goodput with Reno as congestion control.
  Reno achieves the highest goodput in both 3G-UMTS network. However, the 3G-CDMA2000 network gains the highest Goddput in Vegas. In terms of delay, all the networks have highest delay while they use Cubic. As the other types of traffics, TCP Vegas has the highest One-way delay for all networks.

- Stream traffic: In Stream traffic, the networks with higher loss, gain

the higher delay and One-way delay. In addition Reno has the highest delay for each network.

Chapter 8

Appendix

All scripts which measured the data in this thesis , as well as piloting scripts have been uploaded in the repository which can be get from following links.

8.1 Appendix 1: On/Off traffic

on/off traffic (http://bit.ly/1j5Wdav)

8.2 Appendix 2: Bulk traffic

Bulk traffic (http://bit.ly/1lLYc6G)

8.3 Appendix 3: Stream traffic

Stream traffics (http://bit.ly/R5Ex80)

8.4 Appendix 4: Plot Scripts

plot scripts (http://bit.ly/1vBfmLa)

Bibliography

[1] Stefan Alfredsson et al. 'Impact of tcp congestion control on bufferbloat in cellular networks'. In: *Proc. of IEEE WoWMoM13* (2013).

[2] M. Allman, H. Balakrishnan and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042 (Proposed Standard). Internet Engineering Task Force, Jan. 2001. URL: http://www.ietf.org/rfc/rfc3042.txt.

[3] M. Allman, V. Paxson and W. Stevens. *RFC 2581: TCP Congestion Control*. 1999.

[4] Patrik Arlos and Markus Fiedler. 'Influence of the packet size on the one-way delay in 3G networks'. In: *Passive and Active Measurement*. Springer. 2010, pp. 61–70.

[5] Andrea Baiocchi, Angelo Castellani and Francesco Vacirca. 'YeAH-TCP: yet Another Highspeed TCP'. In: *Fifth International Workshop on Protocols for FAST Long-Distance Networks (PFLDnet-07)*. Feb. 2007, pp. 37–42. URL: http://wil.cs.caltech.edu/pfldnet2007/paper/YeAH_TCP.pdf.

[6] A. Bakre and B. R. Badrinath. 'I-TCP: indirect TCP for mobile hosts'. In: *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*. May 1995, pp. 136–143.

[7] Hari Balakrishnan, Venkata N. Padmanabhan and Randy H. Katz. 'The Effects of Asymmetry on TCP Performance'. In: *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '97. Budapest, Hungary: ACM, 1997, pp. 77–89. ISBN: 0-89791-988-2. DOI: 10.1145/262116.262134. URL: http://doi.acm.org/10.1145/262116.262134.

[8] Hari Balakrishnan, Srinivasan Seshan and Randy H. Katz. 'Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks'. In: *Wirel. Netw.* 1.4 (Dec. 1995), pp. 469–481. ISSN: 1022-0038. DOI: 10.1007/BF01985757. URL: http://dx.doi.org/10.1007/BF01985757.

[9] H. Balakrishnan et al. 'A comparison of mechanisms for improving TCP performance over wireless links'. In: *Networking, IEEE/ACM Transactions on* 5.6 (Dec. 1997), pp. 756–769.

[10] E Blanton et al. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. Tech. rep. RFC 6675 (Proposed Standard), 2012.

[11] P. Borgnat et al. 'Seven Years and One Day: Sketching the Evolution of Internet Traffic'. In: *INFOCOM 2009, IEEE*. Apr. 2009, pp. 711–719. DOI: 10.1109/INFCOM.2009.5061979.

[12] L.S. Brakmo and L.L. Peterson. 'TCP Vegas: end to end congestion avoidance on a global Internet'. In: *Selected Areas in Communications, IEEE Journal on* 13.8 (Oct. 1995), pp. 1465–1480.

[13] Kevin Brown and Suresh Singh. 'M-TCP: TCP for Mobile Cellular Networks'. In: *SIGCOMM Comput. Commun. Rev.* 27.5 (Oct. 1997), pp. 19–43. ISSN: 0146-4833. DOI: 10.1145/269790.269794. URL: http://doi.acm.org/10.1145/269790.269794.

[14] R. Caceres and L. Iftode. 'Improving the performance of reliable transport protocols in mobile computing environments'. In: *Selected Areas in Communications, IEEE Journal on* 13.5 (June 1995), pp. 850–857.

[15] Carlo Caini and Rosario Firrincieli. 'Tcp hybla: a tcp enhancement for heterogeneous networks'. In: *INTERNATIONAL JOURNAL OF SATELLITE COMMUNICATIONS AND NETWORKING* 22 (2004).

[16] Claudio Casetti et al. 'TCP Westwood: End-to-end Congestion Control for Wired/Wireless Networks'. In: *Wirel. Netw.* 8.5 (Sept. 2002), pp. 467–479. ISSN: 1022-0038. DOI: 10.1023/A:1016590112381. URL: http://dx.doi.org/10.1023/A:1016590112381.

[17] Mun Choon Chan and Ramachandran Ramjee. 'TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation'. In: *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*. MobiCom '02. Atlanta, Georgia, USA: ACM, 2002, pp. 71–82. ISBN: 1-58113-486-X. DOI: 10.1145/570645.570655. URL: http://doi.acm.org/10.1145/570645.570655.

[18] Cisco. *Voice Over IP - Per Call Bandwidth Consumption*. Feb. 2006. URL: http://bit.ly/RUEjlD.

[19] Mark Claypool et al. 'Characterization by measurement of a CDMA 1x EVDO network'. In: *Proceedings of the 2nd annual international workshop on Wireless internet*. ACM. 2006, p. 2.

[20] A. Dell'Aera, L.A. Grieco and S. Mascolo. 'Linux 2.4 implementation of Westwood+ TCP with rate-halving: a performance evaluation over the Internet'. In: *Communications, 2004 IEEE International Conference on*. Vol. 4. June 2004, 2092–2096 Vol.4. DOI: 10.1109/ICC.2004.1312887.

[21] Thomas Dreibholz. 'The NorNet Testbed: A Platform for Evaluating Multi-Path Transport in the Real-World Internet'. In: *Proceedings of the 87th IETF Meeting*. Berlin/Germany, 30th July 2013. URL: https://simula.no/publications/miscreference.2013-07-30.66541790h06/simula_pdf_file.

[22] Thomas Dreibholz et al. 'Evaluation of a new multipath congestion control scheme using the NetPerfMeter tool-chain'. In: *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*. IEEE. 2011, pp. 1–6.

[23] Hala Elaarag. 'Improving TCP Performance over Mobile Networks'. In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 357–374. ISSN: 0360-0300. DOI: 10.1145/568522.568524. URL: http://doi.acm.org/10.1145/568522.568524.

[24] Ericsson. *The Interim Ericsson Mobility Report - On the pulse of the Networked Society*. Feb. 2014. URL: http://bit.ly/O5FtJ6.

[25] Kevin Fall and Sally Floyd. 'Simulation-based Comparisons of Tahoe, Reno and SACK TCP'. In: *SIGCOMM Comput. Commun. Rev.* 26.3 (July 1996), pp. 5–21. ISSN: 0146-4833. DOI: 10.1145/235160.235162. URL: http://doi.acm.org/10.1145/235160.235162.

[26] Simone Ferlin-Oliveira et al. 'Measuring the QoS Characteristics of Operational 3G Mobile Broadband Networks'. In: *Proceedings of the 4th International Workshop on Protocols and Applications with Multi-Homing Support (PAMS)*. Ed. by IEEE. Victoria, British Columbia/Canada: IEEE, May 2014.

[27] Floyd. *RFC 2582, The New-Reno Modification to TCP's Fast Recovery Algorithm*. Apr. 1999. URL: http://www.ietf.org/rfc/rfc2582.txt.

[28] Sally Floyd et al. 'RFC 2883: An extension to the selective acknowledgement (SACK) option for TCP'. In: *Network Working Group, Internet Engineering Task Force* (2000).

[29] S. Floyd et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. United States, 2000.

[30] A. Ford et al. 'Architectural Guidelines for Multipath TCP Development'. In: *Internet Engineering Task Force (IETF) , Request for Comments* 6182 (Mar. 2011), p. 28.

[31] Cheng Peng Fu and S.C. Liew. 'TCP Veno: TCP enhancement for transmission over wireless access networks'. In: *Selected Areas in Communications, IEEE Journal on* 21.2 (Feb. 2003), pp. 216–228. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.807336.

[32] T. Goff et al. 'Freeze-TCP: a true end-to-end TCP enhancement mechanism for mobile environments'. In: *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. Mar. 2000, 1537–1545 vol.3.

[33] Ernst Gunnar Gran, Thomas Dreibholz and Amund Kvalbein. 'NorNet Core - A multi-homed research testbed'. In: *Computer Networks* 61 (2014). Special issue on Future Internet Testbeds - Part I, pp. 75–87. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.bjp.2013.12.035. URL: http://www.sciencedirect.com/science/article/pii/S1389128613004489.

[34] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: A New TCP-friendly High-speed TCP Variant'. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[35] S. Henna. 'A Throughput Analysis of TCP Variants in Mobile Wireless Networks'. In: *Next Generation Mobile Applications, Services and Technologies, 2009. NGMAST '09. Third International Conference on*. Sept. 2009, pp. 279–284. DOI: 10.1109/NGMAST.2009.71.

[36] J.R. Iyengar, P.D. Amer and R. Stewart. 'Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths'. In: *Networking, IEEE/ACM Transactions on* 14.5 (Oct. 2006), pp. 951–964. ISSN: 1063-6692. DOI: 10.1109/TNET.2006.882843.

[37] V. Jacobson. 'Congestion Avoidance and Control'. In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: 10.1145/52325.52356. URL: http://doi.acm.org/10.1145/52325.52356.

[38] V. Jacobson. 'Modified TCP Congestion Avoidance Algorithm'. In: *end2end-interest mailing list* (30th Apr. 1990).

[39] V. Jacobson, R. Braden and D. Borman. *TCP Extensions for High Performance*. RFC 1323 (Proposed Standard). Internet Engineering Task Force.

[40] Jangeun Jun and M.L. Sichitiu. 'Fairness and QoS in multihop wireless networks'. In: *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*. Vol. 5. Oct. 2003, 2936–2940 Vol.5.

[41] P. Karn and C. Partridge. 'Improving Round-trip Time Estimates in Reliable Transport Protocols'. In: *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*. SIGCOMM '87. Stowe, Vermont, USA: ACM, 1988, pp. 2–7. ISBN: 0-89791-245-4. DOI: 10.1145/55482.55484. URL: http://doi.acm.org/10.1145/55482.55484.

[42] Dominik Kaspar. 'Multipath Aggregation of Heterogenous Access Networks'. PhD thesis. University of Oslo, 2012.

[43] Tom Kelly. 'Scalable TCP: Improving Performance in Highspeed Wide Area Networks'. In: *SIGCOMM Comput. Commun. Rev.* 33.2 (Apr. 2003), pp. 83–91. ISSN: 0146-4833. DOI: 10.1145/956981.956989. URL: http://doi.acm.org/10.1145/956981.956989.

[44] Amit Kumar, Dr Yunfei Liu and Dr Jyotsna Sengupta. 'Divya,"Evolution of Mobile Wireless Communication Networks 1G to 4G"'. In: *International Journal of Electronics and Communication Technology, IJECT* 1.1 (2010).

[45] Aleksandar Kuzmanovic and Edward W. Knightly. 'TCP-LP: Low-priority Service via End-point Congestion Control'. In: *IEEE/ACM Trans. Netw.* 14.4 (Aug. 2006), pp. 739–752. ISSN: 1063-6692. DOI: 10.1109/TNET.2006.879702. URL: http://dx.doi.org/10.1109/TNET.2006.879702.

[46] Amund Kvalbein et al. 'The Nornet Edge platform for mobile broadband measurements'. In: *Computer Networks* 61 (2014). Special issue on Future Internet Testbeds - Part I, pp. 88–101. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.bjp.2013.12.036. URL: http://www.sciencedirect.com/science/article/pii/S1389128613004490.

[47] Yuan-Cheng Lai and Chang-Li Yao. 'The performance comparison between TCP Reno and TCP Vegas'. In: *Parallel and Distributed Systems: Workshops, Seventh International Conference on, 2000*. Oct. 2000, pp. 61–66. DOI: 10.1109/PADSW.2000.884516.

[48] T. V. Lakshman and Upamanyu Madhow. 'The Performance of TCP/IP for Networks with High Bandwidth-delay Products and Random Loss'. In: *IEEE/ACM Trans. Netw.* 5.3 (June 1997), pp. 336–350. ISSN: 1063-6692. DOI: 10.1109/90.611099. URL: http://dx.doi.org/10.1109/90.611099.

[49] F. Lefevre and G. Vivier. 'Understanding TCP's behavior over wireless links'. In: *Communications and Vehicular Technology, 2000. SCVT-200. Symposium on*. 2000, pp. 123–130. DOI: 10.1109/SCVT.2000.923350.

[50] Douglas Leith and Robert Shorten. 'H-TCP: TCP for high-speed and long-distance networks'. In: 2004.

[51] J. Liu and S. Singh. 'ATCP: TCP for mobile ad hoc networks'. In: *Selected Areas in Communications, IEEE Journal on* 19.7 (July 2001), pp. 1300–1315.

[52] Shao Liu, Tamer Basar and R. Srikant. 'TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks.' In: *VALUETOOLS*. Ed. by Luciano Lenzini and Rene L. Cruz. Vol. 180. ACM International Conference Proceeding Series. ACM, 30th Jan. 2007, p. 55. ISBN: 1-59593-504-5. URL: http://dblp.uni-trier.de/db/conf/valuetools/valuetools2006.html#LiuBS06.

[53] Xin Liu et al. 'Experiences in a 3G network: interplay between the wireless channel and applications'. In: *Proceedings of the 14th ACM international conference on Mobile computing and networking*. ACM. 2008, pp. 211–222.

[54] P. Lorenz and P. Dini. *Networking – ICN 2005: 4th International Conference on Networking, Reunion Island, France, April 17-21, 2005, Proceedings*. Lecture Notes in Computer Science / Computer Communication Networks and Telecommunications. Springer, 2005. ISBN: 9783540253389.

[55] Reiner Ludwig and Randy H Katz. 'The Eifel algorithm: making TCP robust against spurious retransmissions'. In: *ACM SIGCOMM Computer Communication Review* 30.1 (2000), pp. 30–36.

[56] Matthew Mathis and Jamshid Mahdavi. 'Forward acknowledgement: Refining TCP congestion control'. In: *ACM SIGCOMM Computer Communication Review* 26.4 (1996), pp. 281–291.

[57] Matt Mathis et al. *TCP selective acknowledgment options*. Tech. rep. RFc 2018, October, 1996.

[58] Jeonghoon Mo et al. 'Analysis and comparison of TCP Reno and Vegas'. In: *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. Mar. 1999, 1556–1563 vol.3. DOI: 10.1109/INFCOM.1999.752178.

[59] Thi-Ha Nguyen et al. 'An improvement of TCP performance over wireless networks'. In: *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*. July 2013, pp. 214–219.

[60] V. Paxson and M. Allman. *Computing TCP's Retransmission Timer*. RFC 2988 (Proposed Standard). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2988.txt.

[61] J. Postel. *Transmission Control Protocol*. RFC 793 (Standard). Updated by RFCs 1122, 3168. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[62] J. Postel. *User Datagram Protocol*. RFC 768. Internet Engineering Task Force, Aug. 1980, p. 3. URL: http://www.rfc-editor.org/rfc/rfc768.txt.

[63] Peter J. Rousseeuw, Ida Ruts and John W. Tukey. 'The Bagplot: A Bivariate Boxplot'. In: *The American Statistician* 53.4 (1999), pp. 382–387. DOI: 10.1080/00031305.1999.10474494. eprint: http://www.tandfonline.com/doi/pdf/10.1080/00031305.1999.10474494. URL: http://www.tandfonline.com/doi/abs/10.1080/00031305.1999.10474494.

[64] B. Sardar and D. Saha. 'A survey of tcp enhancements for last-hop wireless networks'. In: *Communications Surveys Tutorials, IEEE* 8.3 (rd 2006), pp. 20–34. ISSN: 1553-877X. DOI: 10.1109/COMST.2006.253273.

[65] Pasi Sarolahti and Alexey Kuznetsov. 'Congestion Control in Linux TCP.' In: *USENIX Annual Technical Conference, FREENIX Track*. 2002, pp. 49–62.

[66] P Sarolahti et al. *RFC 5682-Forward RTO-Recovery (F-RTO)*. 2009.

[67] Wee Lum Tan, Fung Lam and Wing Cheong Lau. 'An empirical study on the capacity and performance of 3g networks'. In: *Mobile Computing, IEEE Transactions on* 7.6 (2008), pp. 737–750.

[68] Ye Tian, K. Xu and N. Ansari. 'TCP in wireless environments: problems and solutions'. In: *Communications Magazine, IEEE* 43.3 (Mar. 2005), S27–S32.

[69] V. Tsaoussidis and H. Badr. 'TCP-probing: towards an error control schema with energy and throughput performance gains'. In: *Network Protocols, 2000. Proceedings. 2000 International Conference on*. 2000, pp. 12–21.

[70] Nitin Vaidya et al. *Delayed Duplicate Acknowledgements: A TCP-Unaware Approach to Improve Performance of TCP over Wireless*. 1999.

[71] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated*. Addison-Wesley Professional Computing Series v. 2. Pearson Education, 1995. ISBN: 9780321617644.

[72] E.H.-K. Wu and Mei-Zhen Chen. 'JTCP: jitter-based TCP for heterogeneous wireless networks'. In: *Selected Areas in Communications, IEEE Journal on* 22.4 (May 2004), pp. 757–766.

[73] Lisong Xu, K. Harfoush and Injong Rhee. 'Binary increase congestion control (BIC) for fast long-distance networks'. In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. Vol. 4. Mar. 2004, 2514–2524 vol.4. DOI: 10.1109/INFCOM.2004.1354672.

[74] G. Xylomenos and G.C. Polyzos. 'TCP and UDP performance over a wireless LAN'. In: *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 2. Mar. 1999, 439–446 vol.2. DOI: 10.1109/INFCOM.1999.751376.

[75] P. Yang et al. *TCP Congestion Avoidance Algorithm Identification*. 2013. DOI: 10.1109/TNET.2013.2278271.