

2014

TCP FTAT (Fast Transmit Adaptive Transmission): a New End-To-End Congestion Control Algorithm

Mohammed Ahmed Melegy Mohammed Afifi
Cleveland State University

Follow this and additional works at: <https://engagedscholarship.csuohio.edu/etdarchive>

 Part of the [Electrical and Computer Engineering Commons](#)

How does access to this work benefit you? Let us know!

Recommended Citation

Afifi, Mohammed Ahmed Melegy Mohammed, "TCP FTAT (Fast Transmit Adaptive Transmission): a New End-To-End Congestion Control Algorithm" (2014). *ETD Archive*. 730.

<https://engagedscholarship.csuohio.edu/etdarchive/730>

This Thesis is brought to you for free and open access by EngagedScholarship@CSU. It has been accepted for inclusion in ETD Archive by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.

TCP FTAT (Fast Transmit Adaptive Transmission): A NEW END-TO-END CONGESTION CONTROL ALGORITHM

MOHAMMED AHMED MELEGY MOHAMMED AFIFI

Bachelor of Electronics Engineering and Technology
World College
July 2011

submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY
December 2014

We hereby approve this thesis

For

MOHAMMED AHMED MELEGY MOHAMMED AFIFI

Candidate for the Master of Science in Electrical Engineering degree

For the Department of

Electrical and Computer Engineering

And

CLEVELAND STATE UNIVERSITY'S

College of Graduate Studies by

Dr. Nigamanth Sridhar, Committee Chair

Department & Date

Dr. Chansu Yu, Committee Member

Department & Date

Dr. Pong Chu, Committee Member

Department & Date

May 7, 2014

Student's Date of Defense

To my beloved parents...

ACKNOWLEDGEMENTS

I would like to thank all those who gave me the opportunity to complete my thesis work. I would like to thank my advisor, Professor Nigamanth Sridhar, for all the guidance and the support throughout my thesis work. I would also like to thank the Department of Electrical and Computer Engineering, of which I have had the pleasure being a student for the past two years.

TCP FTAT (Fast Transmit Adaptive Transmission): A NEW END-TO-END CONGESTION CONTROL ALGORITHM

MOHAMMED AHMED MELEGY MOHAMMED AFIFI

ABSTRACT

Congestion Control in TCP is the algorithm that controls allocation of network resources for a number of competing users sharing a network. The nature of computer networks, which can be described from the TCP protocol perspective as unknown resources for unknown traffic of users, means that the functionality of the congestion control algorithm in TCP requires explicit feedback from the network on which it operates. Unfortunately this is not the way it works with TCP, as one of the fundamental principles of the TCP protocol is to be end-to-end, in order to be able to operate on any network, which can consist of hundreds of routers and hundreds of links with varying bandwidth and capacities. This fact requires the Congestion Control algorithm to be adaptive by nature, to adapt to the network environment under any given circumstances and to obtain the required feedback implicitly through observation and measurements. In this thesis we propose a new TCP end-to-end congestion control algorithm that provides performance improvements over existing TCP congestion control algorithms in computer networks in general, and an even greater improvement in wireless and/or high bandwidth-delay product networks.

Table of Contents

ABSTRACT	v
Table of Contents	vi
List of Figures	x
LIST OF TABLES	xii
CHAPTER I	1
Introduction and motivation	1
1.2. TCP NewReno currently is not suitable for today’s networks	2
1.3. Solutions specific for Wireless Networks	4
1.3.1. Indirect TCP (I-TCP)	5
1.3.2. Snoop protocol	5
1.3.3. Multicast TCP (M-TCP)	6
1.3.4. Explicit Congestion Notification (ECN)	6
1.4. The Thesis	7
1.5. Statement of Purpose	8
CHAPTER II	9
NewReno and Westwood	9
2.1. NewReno	9
2.1.1. Slow-Start	9
2.1.2. Congestion Avoidance	10

2.1.3.	Fast Retransmit and Fast Recovery	10
2.2.	TCP Westwood and Westwood+	12
2.2.1.	Congestion Window Update in Westwood	13
2.2.2.	Westwood Bandwidth Estimation Mechanism	14
2.2.3.	Westwood Packet Counting Procedure	16
2.2.4.	Westwood+	17
CHAPTER III		19
FTAT – A New Congestion Control Algorithm		19
3.1.	The Problem	19
3.2.	Solution	20
3.3.	End-to-End Loss Scenario	21
3.3.1.	Ideal Congestion Scenario	21
3.3.2.	Ideal Wireless Segments Loss.....	22
3.3.3.	Actual Scenario.....	23
3.4.	FTAT Approach.....	24
3.5.	TCP FTAT Congestion Control.....	26
3.5.1.	Initial Congestion Window	28
3.5.2.	FTAT Algorithm	29
CHAPTER IV		31
Implementation of FTAT in ns-3		31

4.1.	The Network Simulator – 3	31
4.2.	The Implementation of TCP in ns-3	31
4.3.	Implementation of FTAT in ns-3	32
CHAPTER V		37
TCP FTAT Linux-Stack Implementation		37
5.1.	Introduction	37
5.2.	TCP Congestion Control in Linux	37
5.3.	FTAT Implementation in Linux TCP Stack	43
CHAPTER VI.....		46
A Mathematical Model of TCP FTAT		46
Theorem: A simplified steady state throughput of the FTAT algorithm		46
Corollary: The FTAT congestion control is stable ($TFTAT \leq B$)		48
CHAPTER VII		49
Evaluation based comparison of TCP-NewReno, TCP-Westwood+, and TCP-FTAT using ns-3		49
7.1.	Topology One	49
7.2.	Topology Two	56
7.3.	Topology Three	62
7.4.	Fourth Topology: Two-way Geo Satellite Scenario.....	67
CHAPTER VIII.....		74
Evaluation and Comparison of Different Congestion Control Algorithms of Linux		

Stack Using DCE Cradle (Direct Code Execution Cradle)	74
8.1. Introduction	74
8.2. Simulation and Comparisons	75
8.2.1. Topology One: One-way Geo Satellite Scenario	75
8.2.2. Topology Two	81
8.2.3. Topology Three	83
8.2.4. Fourth Topology	85
CHAPTER IX	88
Conclusion	88
BIBLIOGRAPHY	91

List of Figures

Figure 1: FSM Description of TCP Congestion Control [reproduced from 16].	12
Figure 2: FSM description of Westwood [produced from 18, 19]	18
Figure 3: Black Box Principle in the Presence of Congestion	21
Figure 4: Black Box Principle in the Presence of Random Loss	22
Figure 5: Black Box Principle in the Presence of Congestion and Random Loss	23
Figure 6: Adaptive Transmission Effect	25
Figure 7: NewReno Congestion Window Pattern	27
Figure 8: FSM description of FTAT	30
Figure 9: TCP Implementation in ns-3 including FTAT	34
Figure 10: Classes interaction in Linux	40
Figure 11: TCP function interaction in Linux	42
Figure 12: First Topology	49
Figure 13: cwnd graphs for Topology one	50
Figure 14: RTT graphs for topology two	52
Figure 15: Sequence number topology one	53
Figure 16: Throughput and Goodput topology one	55
Figure 17: Topology two	56
Figure 18: cwnd graphs topology two	57
Figure 19: RTT graphs for topology two	59
Figure 20: Sequence number topology two	59
Figure 21: Throughput and Goodput for topology two	60
Figure 22: Third Topology	62

Figure 23: cwnd topology three	63
Figure 24: RTT graphs topology three.....	65
Figure 25: Throughput topology three	66
Figure 26: Topology four	67
Figure 27: cwnd topology four.....	68
Figure 28: RTT topology four.....	70
Figure 29: Sequence number topology four.....	71
Figure 30: Throughput and goodput topology four.....	72
Figure 31: Topology one.....	75
Figure 32: cwnd topology one.....	78
Figure 33: Sequence number topology one.....	79
Figure 34: Highest sequence number	79
Figure 35: Throughput topology one	80
Figure 36: Topology two.....	81
Figure 37: Throughput topology two	82
Figure 38: Topology Three	83
Figure 39: Throughput topology three	84
Figure 40: Topology Four	85
Figure 41: Throughput topology four.....	86

LIST OF TABLES

Table 1: Slow Start.....	20
Table 2: Congestion Avoidance.....	21
Table 3: Fast Recovery	22
Table 4: Westwood window update.....	25
Table 5: Bandwidth Sampling.....	26
Table 6: Westwood packets counting procedure.....	27
Table 7: Westwood Bandwidth Sampling.....	29

CHAPTER I

Introduction and motivation

1.1. Introduction

TCP Congestion Control has gone through many improvements and enhancements over the past 26 years, since Van Jacobson proposed the original Tahoe algorithm in 1988 [1]. One of the most deployed algorithms is TCP NewReno [2], which is an improvement over the original TCP Tahoe. The first transition was from TCP Tahoe to TCP Reno through adding a new algorithm called Fast Recovery in 1990 by Van Jacobson [3]. The second transition was by Sally Floyd and T. Henderson in 1999, through enhancing the Fast Recovery algorithm to recover from multiple losses in the same window [2]. Since that time, wired networks have advanced and congestion became almost the only cause for timeout and data loss in wired networks. At the same time, wireless technology has advanced and wireless networks have been deployed rapidly, which caused the radio channel errors to be the main source of packets loss after congestion in wireless networks. This evolution has required a change in the way the congestion signal should be handled.

1.2. TCP NewReno currently is not suitable for today's networks

- a. High bandwidth-delay product networks that are currently in increased deployment, require a rate of increase in the *congestion window (cwnd)* that is more than a linear increase of one *Maximum Segment Size (MSS)*, every *round-trip time (RTT)* to grab its share of the network bandwidth, which is due to the high RTT that encountered in such networks, which are hundreds of milliseconds.

 - b. At the start-up phase, *cwnd* starts with a maximum of 4380 Bytes [4] and increases slowly, which takes a long time to gain a proper window size and hence good throughput, and yet a single packet loss identified by three duplicate acknowledgments will reduce the *cwnd* to half of the current value.

 - c. There are no obvious differences between packet loss caused by congestion and loss caused by a wireless connection, and hence all losses are assumed to be congestion and handled in the same way, which degrades the overall throughput for a given connection operating over hybrid network consists of wireless as well as wired networks.
-

Since the original congestion control algorithm by Van Jacobson, many proposals have been introduced to address these issues. Some of these algorithms which have been studied for many years by researchers are Westwood, Vegas, Veno, and SACK.

Westwood is a modification of the NewReno algorithm in the sender-side, which is less sensitive to random loss in the wireless environments than NewReno due to its behavior when a loss is detected [5]. Westwood reacts to a segment loss by adjusting the *cwnd* to an estimated value of the network's available bandwidth.

Westwood+ [6] introduced a modified bandwidth measurement procedure different from the one used in Westwood. The bandwidth estimation procedure used in Westwood+ collects a sample every RTT instead of every acknowledgment. This reduces the effect of acknowledgment compression. The simulation results presented in this thesis show that Westwood+ suffers from performance degradation when operating under reverse traffic.

Vegas is another end-to-end approach to congestion control, which bases its link bandwidth estimation process on the RTT [7]. Vegas measures the RTT, then performs a comparison between the *actual* rate of sending, computed as

$(\text{Congestion Window} / \text{measured RTT})$ to the *expected* rate of sending using the minimum measured RTT computed, as $(\text{Congestion Window} / \text{minimum RTT})$ [7]. After computing the difference between the rate of sending and the expected rate of sending, three scenarios could happen:

1. The Congestion Window is increased additively, if the computed difference reveals that it is less than threshold α .

2. The Congestion Window is decreased additively, if the computed difference reveals that it is larger than threshold β .
3. The Congestion Window is kept the same; if the computed difference is less than β and larger than α .

TCP Vegas operates on a principle of congestion prevention, which tries to prevent congestion instead of dealing with it after it happens. Studies [8] show that it yields better throughput than Reno in specific scenarios, but in other studies such as [9] it has been shown that TCP Vegas, when competing with other congestion control algorithms such as Reno that tries to achieve the network capacity in systematic way, cannot allocate its share of the network bandwidth.

1.3. Solutions specific for Wireless Networks

Because of the problem of the random loss, and the stability of Additive-Increase, Multiplicative-Decrease (AIMD) algorithms such as NewReno in wired networks, calls have been introduced for new approaches for wireless networks and proposals have been introduced as a result [10]. The approaches that are designed for wireless or hybrid networks specifically, which usually deploy a split mechanism or a modification to the TCP structure, usually does not follow the end-to-end principle [6]. The split approach splits a hybrid network into a wired portion and a wireless portion. In such a case, the wired portion operates by using a conventional congestion control algorithm, which is usually an AIMD approach such as NewReno. Whereas the wireless network access point operates by using protocols that manage the acknowledgment returned from the wireless network. Some approaches that employ the split-connection semantic are Indirect TCP (I-

TCP) [11], Snoop protocol [14], Multicast TCP (M-TCP) [12], and Explicit Congestion Notification (ECN) [13]. Each of these mechanisms will be discussed in the next sections. Of course the problem of the high bandwidth-delay product of today's networks was not a big concern when these approaches was proposed, which makes them special solutions specific for wireless networks. The next subsections discusses these approaches in more details.

1.3.1. Indirect TCP (I-TCP)

The Indirect TCP (I-TCP) is one of the approaches specific for wireless networks, in which a proxy is inserted between the wired network and the wireless network to manage the connection, and the wireless network operates using a modified TCP congestion control algorithm.

1.3.2. Snoop protocol

Another approach is the Snoop protocol, which can be considered as one of the most successful approaches of these different solutions [15]. TCP Westwood provided 380% improvement over NewReno, while in the same environment, Snoop provided a 400% improvement over NewReno. The Snoop protocol is based between the wired network and the wireless connection. Every packet sent from the wired network to the wireless network is cached at the snoop base. When an acknowledgment is received from the wireless connection, snoop checks for duplicate acknowledgment, if there are duplicate acknowledgment, snoop retransmits the reported lost segment by the duplicate acknowledgment cached packets, and the duplicate acknowledgment is held at the snoop

base. If the retransmission is successful, Snoop will resume the transmission as normal; otherwise snoop sends the duplicate acknowledgment through the wired connection to the sender implicitly reporting congestion.

1.3.3. Multicast TCP (M-TCP)

Multicast TCP (M-TCP) is another approach to deal with wired/wireless connections and specifically the wireless links that have low bit rate. M-TCP operates by splitting the nodes connected through the wired connection FH (fixed-host) from the nodes connected through the wireless connection MH (mobile-host) by a SH (supervisor-host). The FH operates using the standard TCP congestion control, and the MH operates using a special version of TCP. The main purpose of the SH is to manage the communication between the FH and the MH, as when the FH sends a packet to the MH, the SH receives it first and forwards it to the MH. If the MH stops responding, the SH sends an acknowledgment to the FH stating a receiver window size of zero. At that time the FH sends a *probe packet* to the end node (MH), the SH receives the probe packet and responds back with a receiver window size of zero. This process ends when the MH starts responding, at that time the connection resume normally.

1.3.4. Explicit Congestion Notification (ECN)

Explicit Congestion Notification (ECN) operates by reserving two bits in the IP header and two bits in the TCP header for ECN notification. When there is congestion in the network, these bits are set to true, which in turn alerts the receiver that there is congestion and the receiver responds with an acknowledgment with the two bits set to

true. When the sender receives the acknowledgment from the receiver, the sender reacts to the congestion by reducing the *cwnd* and thus the sending rate. If the sender discovers a lost segment and these two bits were set to false, the sender knows that the segment lost was due to wireless link errors and not due to congestion and as a consequence, the sender does not reduce its sending rate. While in this TCP congestion control approach, the loss cause can be identified precisely, ECN requires changes to every node and device involved in the communication process between the two end nodes.

1.4. The Thesis

We defend the following thesis:

A new end-to-end TCP congestion control algorithm that addresses the difficulties faced by the current TCP congestion control; namely the initial throughput, operating over wireless or hybrid networks and operating over large bandwidth-delay product networks. The proposed algorithm does not take the conventional congestion signal (duplicate acknowledgment) as guaranteed sign of congestion, instead, it employ a new approach in testing the cause of the loss to determine the actual network capacity, and as a result does not degrade the throughput due to false congestion signal.

In this thesis, our focus will be on NewReno as it was the dominant congestion control for many years, and Westwood+ because it is one of the most successful end-to-end approaches to congestion control that addresses the random loss issue and shares with FTAT the same principle of using the returning acknowledgment as a feedback to estimate the network capacity, also the comparison of the proposed algorithm will cover all of the Linux implemented congestion control algorithms.

1.5. Statement of Purpose

The function of TCP Congestion Control is to limit the rate of sending when the End-to-End path indicates congestion and to allow the expansion of the *cwnd* to grab its share of the network resources, when there is no indication of congestion.

In the early days of the Internet, the only concern when designing and implementing the congestion control of TCP was to avoid congestion as much as possible, and to deal with congestion when it occurred. Other factors affecting the performance of TCP in today's networks were not, at that time real concerns. With today's wide deployment of wireless technologies, high-speed networks and the high proportion of applications on the Internet, which consist of small amount of data that require throughput at the start-up phase, TCP with its current congestion control algorithm is no longer a suitable standard for all networks. From that point of view a real need for contributions of proposals to the Congestion Control of TCP that address the challenges faced by TCP in today's networks and to complies with the End-to-End semantic of TCP is vital.

CHAPTER II

NewReno and Westwood

2.1. NewReno

2.1.1. Slow-Start

The NewReno algorithm consists of four sub algorithms, which are: Slow-Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. The NewReno congestion window (*cwnd*) starts as minimum of one segment and a maximum of four segments, it increases exponentially by one segment on each successful delivered segment to the destination, indicated by a received acknowledgement at the sender side. The window continues to grow until one of two cases takes place: either the capacity of the network is hit, and in that case the congestion window returns to one, or the Slow Start threshold (*ssthresh*) is achieved, and in that case the Congestion Avoidance starts. The Congestion Window gains the doubles each RTT.

Table 1: Slow Start

<i>Begins</i>	$cwnd \leq ssthresh$
<i>Every new Acknowledgment</i>	$cwnd += MSS$
<i>cwnd gain every RTT:</i>	$cwnd = 2 \times cwnd$
<i>Congestion Avoidance starts</i>	$cwnd >= ssthresh$

2.1.2. Congestion Avoidance

The Congestion Avoidance state starts when the congestion window has reached the Slow-Start threshold, and in that case the congestion window increases slowly to prevent a possible congestion. The congestion window increases by one MSS every RTT or $(MSS \times MSS / cwnd)$ per each new acknowledgement.

A note here that on the first RTT, the *ssthresh* might not reflect the actual network capacity at all, and as a result the congestion avoidance phase starts and the congestion window increases very slowly while it should increase rapidly to achieve the fair share capacity of a high bandwidth network, and as a result the stability of a network is achieved but no adequate throughput is gained.

Table 2: Congestion Avoidance

<i>Starts</i>	$cwnd \geq ssthresh$
<i>Every new Acknowledgment</i>	$cwnd += MSS \times MSS / cwnd$
<i>cwnd gain every RTT:</i>	$cwnd += 1 \times MSS$

2.1.3. Fast Retransmit and Fast Recovery

If a packet loss is identified by three duplicate acknowledgments after the original acknowledgment, the Fast Retransmit starts. In the Fast Retransmit phase, the sequence number of the highest transmitted packet is recorded in a variable called *recover*. The *ssthresh* is set as in the event of retransmit time-out, to the maximum of half the flight-size and two MSS. The *cwnd* is set to the

$ssthresh + 3 \times MSS$ to compensate the available bandwidth indicated by the arrival of the three packets to the receiver indicated by three duplicate acknowledgment. The lost segment is then retransmitted.

In NewReno, the new acknowledgment after a duplicate acknowledgment in the Fast Recovery phase could refer to full acknowledgment or partial acknowledgment. A full acknowledgment is the new acknowledgment that acknowledges all of the transmitted data packets, while a partial acknowledgment is the new acknowledgment that acknowledges only some of the previous transmitted data packets.

In case of a full acknowledgment, the $cwnd$ is set to either the minimum of ($flight-size + 1 \times MSS$) or $ssthresh$, and the Fast Recovery is exited. In the case of a partial acknowledgment, the first sequence number in the $cwnd$ which has not been acknowledged yet is retransmitted, the $cwnd$ is deflated back to the amount of data that has been acknowledged plus one MSS, and one new packet is transmitted [2].

Table 3: Fast Recovery

<i>recover variable:</i>	<i>Highest transmitted packet sequence number</i>
<i>flight-size:</i>	<i>Minimum (rwnd, cwnd)</i>
<i>ssthresh :</i>	<i>Maximum (flight-size/2, 2)</i>
<i>cwnd:</i>	<i>$ssthresh + 3 \times MSS$</i>
<i>Lost Segment Retransmitted</i>	
<i>Partial acknowledgment:</i>	<i>Send highest sequence number not acknowledged</i>
<i>cwnd:</i>	<i>amount of data acknowledged + 1 × MSS</i>
<i>New Packet is transmitted</i>	

<i>full acknowledgment:</i>	
<i>cwnd:</i>	<i>Minimum (ssthresh, flight-size + 1 × MSS)</i>
<i>Exit Fast Recovery, resume Congestion Avoidance.</i>	

Reno

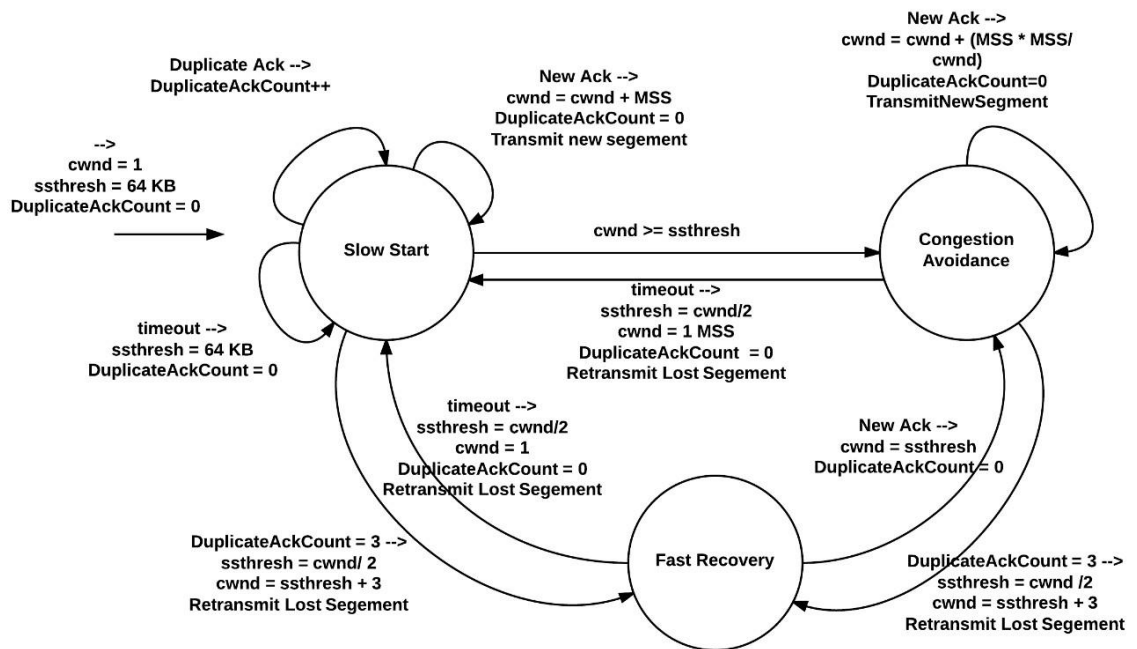


Figure 1: FSM Description of TCP Congestion Control [reproduced from 16].

2.2. TCP Westwood and Westwood+

Westwood is a congestion control algorithm that was designed to address the random loss issue in wireless networks, and is a modification of NewReno that uses a different procedure when a loss is detected. In the Slow-Start and Congestion Avoidance phases, Westwood increases the *cwnd* the same way as NewReno, one MSS every new acknowledgment, and one MSS every RTT in the Congestion Avoidance phase.

The main difference between NewReno and Westwood is seen when a three duplicate

acknowledgment are received, or retransmission time-out occurs. Westwood employs a novel bandwidth estimation mechanism that is used to set the *cwnd* and *ssthresh* upon receiving three duplicate acknowledgments or encountering a retransmission time-out. After setting the new values for *ssthresh* and *cwnd*, the algorithm performs normal Fast Retransmit and Fast Recovery as in NewReno.

2.2.1. Congestion Window Update in Westwood

Westwood relies on the feedback of the returning acknowledgments to estimate the network bandwidth. After a loss is acknowledged by way three duplicate acknowledgments, the *ssthresh* and *cwnd* are adjusted according to the bandwidth measured at the time of congestion multiplied by the minimum RTT observed during the connection; the result is then divided by MSS. After the *ssthresh* is set, the *cwnd* is compared to the value of *ssthresh*, and if the *cwnd* value is greater than *ssthresh*, the *cwnd* is set equal to the *ssthresh*, and the gain rate of *cwnd* is the same as in the

Congestion Avoidance phase. Otherwise, no change is made to the *cwnd* value, and the algorithm resumes in the Slow Start phase [5].

If Westwood detects the loss by a retransmission time-out, the *ssthresh* and the *cwnd* are set in a different way. First *ssthresh* is set in the same manner, and then *ssthresh* is checked, if the value is less than two, *ssthresh* is set equal to two. The *cwnd* is set in the same way as in NewReno after retransmission time-out.

After Three Duplicate Acknowledgment:

Table 4: Westwood window update

<i>ssthresh</i> :	<i>Maximum (Measured Bandwidth × minimum RTT/ Segment Size, 2)</i>
<i>cwnd</i> > <i>ssthresh</i> :	<i>cwnd = ssthresh (Congestion Avoidance)</i>
<i>cwnd</i> ≤ <i>ssthresh</i> :	<i>(No Change, Slow Start)</i>

After Retransmission time-out:

<i>ssthresh</i> :	<i>Maximum ((Measured Bandwidth × minimum RTT/ Segment Size), 2)</i>
<i>Ssthresh</i> < 2:	<i>ssthresh = 2</i>
<i>cwnd</i> :	<i>cwnd = 1 (Slow Start)</i>

2.2.2. Westwood Bandwidth Estimation Mechanism

The available bandwidth in the network is calculated as the number of data bytes acknowledged during the recent received acknowledgment divided by the difference in the time between the most recent acknowledgment and the previous acknowledgment.

Westwood measures the bandwidth after each acknowledgment is received. When a loss happens, the bandwidth sample is processed into a low-pass filter to obtain the low- frequency average component of the sample.

Table 5: Bandwidth Sampling

<i>Bandwidth Sample</i> (<i>Bk</i>):	$Dk / \Delta k$
<p>Where</p> <p>Dk : Data acknowledged in Bytes (Number of acknowledged segments \times Segment Size)</p> <p>Δk : Time of The Received Acknowledgment – Time of The Previous Acknowledgment</p>	

The filtering process is achieved by Tustin approximation [17, 5] is as follow:

$\hat{B}K = \alpha K \times \hat{B}k-1 + (1 - \alpha K) (Bk + Bk-1)/2$
<p>Where</p> <p>$\hat{B}K$: The Filtered Bandwidth at time ($t = tk$)</p> <p>$\alpha K = (2\tau - \Delta k)/(2\tau + \Delta k)$, where $\Delta k = tk - tk-1$ and $1/\tau$ is the filter cutoff frequency</p> <p>$\hat{B}k-1$: The last filtered bandwidth sample</p> <p>Bk: The recent bandwidth sample</p> <p>$Bk-1$: The pervious bandwidth sample</p>

The coefficient αK has been chosen to be dependent on the inter-arrival time Δk .

The relationship between the inter-arrival time Δk and the coefficient αK is inversely proportion. So when the inter-arrival time increases, the value of the coefficient decreases and hence the significance of the last filtered sample ($\hat{B}k-1$) decreases. On the other hand, when the inter-arrival time decreases the

significance of the last filtered sample increases.

2.2.3. Westwood Packet Counting Procedure

Westwood uses a very accurate counting procedure for data bytes acknowledged. The counting procedure takes into count the delayed and cumulative acknowledgements.

Table 6: Westwood packets counting procedure

<i>cumul_ack:</i>	<i>Current Acknowledgement Sequence – Last Acknowledgement Sequence</i>
<i>cumul_ack == 0</i>	<i>accounted_for + 1; cumul_ack = 1; (Duplicate Acknowledgment)</i>
<i>cumul_ack > 1 && accounted_for >= cumul_ack</i>	<i>accounted_for = (accounted_for - cumul_ack), cumul_ack = 1 (Delayed Acknowledgment)</i>
<i>cumul_ack > 1 && accounted_for < cumul_ack</i>	<i>cumul_ack = cumul_ack - accounted_for; accounted_for = 0 (New Acknowledgment)</i>
<i>Last Acknowledgment Sequence = Current Acknowledgement Sequence (Update Acknowledgment Sequence Number)</i>	
<i>acked:</i>	<i>cumul_ack</i>
<i>return (acked)</i>	

Where

cumul_ack: The Number of Acknowledged Segments accounted_for:

The Number of Duplicate Acknowledgment

acked : Number of Acknowledged Segments Reported by Current Received Acknowledgment

2.2.4. Westwood+

Westwood+ is a further refinement of Westwood, with the key improvement occurring in the bandwidth measurement procedure. The available bandwidth measurement of Westwood+ relies on the acknowledged data bytes during one RTT period, which provides a better measurement of the available bandwidth and eliminates the dependency of the acknowledgment inter-arrival times.

Bandwidth Sampling:

<i>Bandwidth Sample</i> (Bk):	$Dk / \Delta k$
----------------------------------	-----------------

Where

Dk : Data acknowledged in Bytes (Number of acknowledged segments \times Segment Size)

Δk : The RTT of The Computed Sample

The time-invariant filter proposed in Westwood+ is a modified version Westwood time-variant filter [15, 18]:

$$\hat{B}K = \alpha K \times \hat{B}k-1 + (1 - \alpha K) \times Bk$$

Where

$\hat{B}K$: The Filtered Bandwidth at time ($t = tk$)

$\alpha K = 0.9$

$\hat{B}k-1$: The last filtered bandwidth sample

Bk : The recent bandwidth sample

Table 7: Westwood Bandwidth Sampling

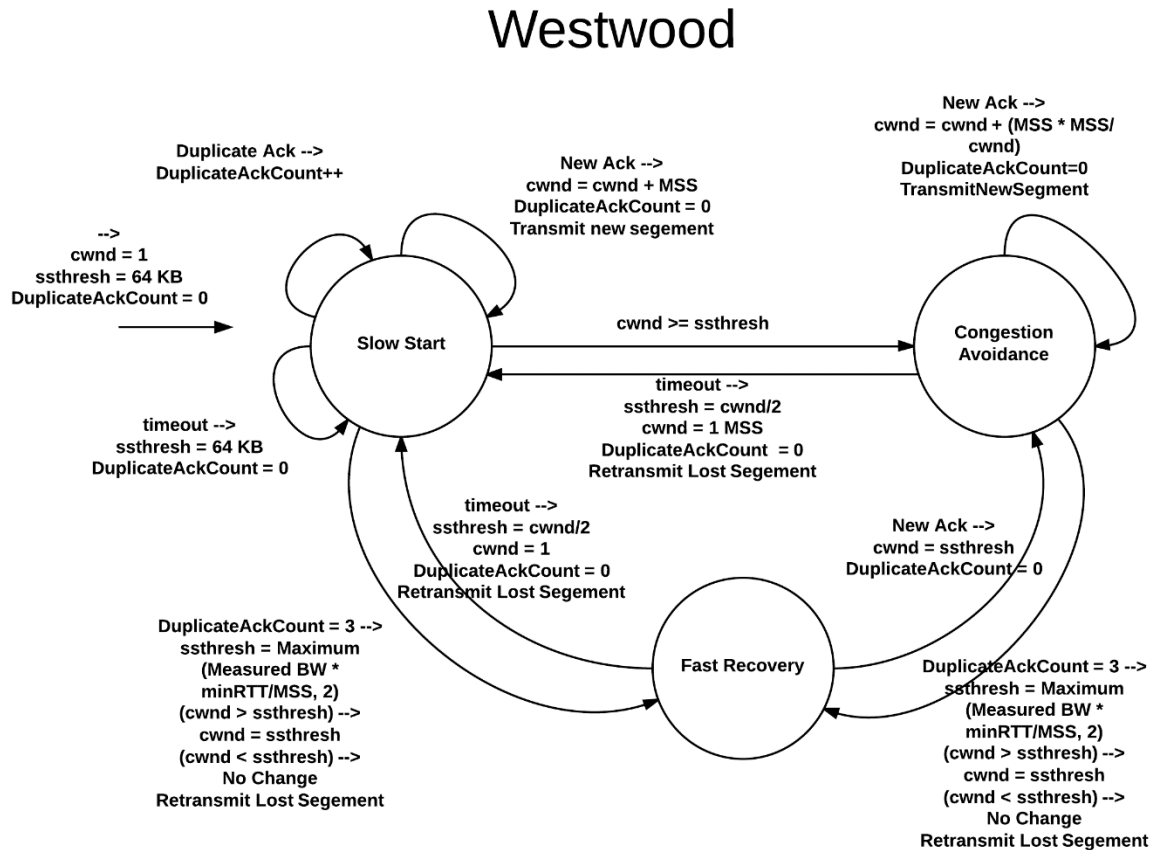


Figure 2: FSM description of Westwood [produced from 18, 19]

CHAPTER III

FTAT – A New Congestion Control Algorithm

One of the most fundamental principles of the Transmission Control Protocol (TCP) is that the congestion control must be End-to-End. In other words, there must be no explicit feedback from the network between the two end-systems [5]. This design principle of TCP allows the connection to be reliable no matter what kind of networks it operates on, and the kinds of failures that can be encountered in the intermediate nodes. Therefore, any information about the network needs to be obtained using measurements and observations, while treating the network as a “black box”.

3.1. The Problem

The problem that researchers have been studying for many years is how to distinguish data loss caused by radio links (random loss), from that caused by congestion. This distinction is difficult to pin down, as the data bytes lost during a connection due to radio links are random and suggests no specific systematic way that can be traced and differed than that of congestion. As well, there are other important attributes involved in the reliable communication of the TCP connection such as network stability, fairness of shared network bandwidth among nodes sharing a network operating over a TCP connection, and inter-protocol friendliness of different kinds of TCP implementations. Some researchers suggest an explicit notification from some network devices such as routers to determine the connection type, and as a result handle the loss in a proper way

[20]. Other researchers have proposed installing proxies between the radio links and the wired links to isolate each connection from the other and hence handle losses in a proper way [13, 11, 21]. Finally there is the end-to-end solution, which complies with the TCP principles as an End-to-End reliable Transmission Control Protocol.

The primary reason to have to distinguish data loss from random loss from that caused by congestion is that the data loss in the two cases needs different treatment. The loss caused by congestion requires immediate action from the TCP sender to reduce the rate of segments transmitted to the network in order to avoid congestion collapse, while the random loss should not have any effect on the rate of sending as the loss cause is not urgent (or repeatable). In reality, however, the original TCP (Reno) does not have a mechanism to distinguish between the random loss from congestion loss, as a result any segment loss is considered congestion and the *cwnd* is cut to half if the loss signal is three duplicate acknowledgments, and reduced to one segment if the loss signal is a retransmission time-out.

3.2. Solution

The proposed solution is whenever a loss signal has been activated, the network capacity is “tested” to measure the reality of the loss cause. One of the ways this “testing” can be performed is by sending a defined amount of data, and observing the received data at the end-node in a specific period of time, then adjusting the *cwnd* accordingly. In correspondence to the loss scenario, duplicating the same environment with the same attributes where the loss occurred reveals the cause of the loss; duplicating the same

environment in terms of one RTT and the *cwnd* size. In the next section, a visualization of loss scenarios is presented.

3.3. End-to-End Loss Scenario

3.3.1. Ideal Congestion Scenario

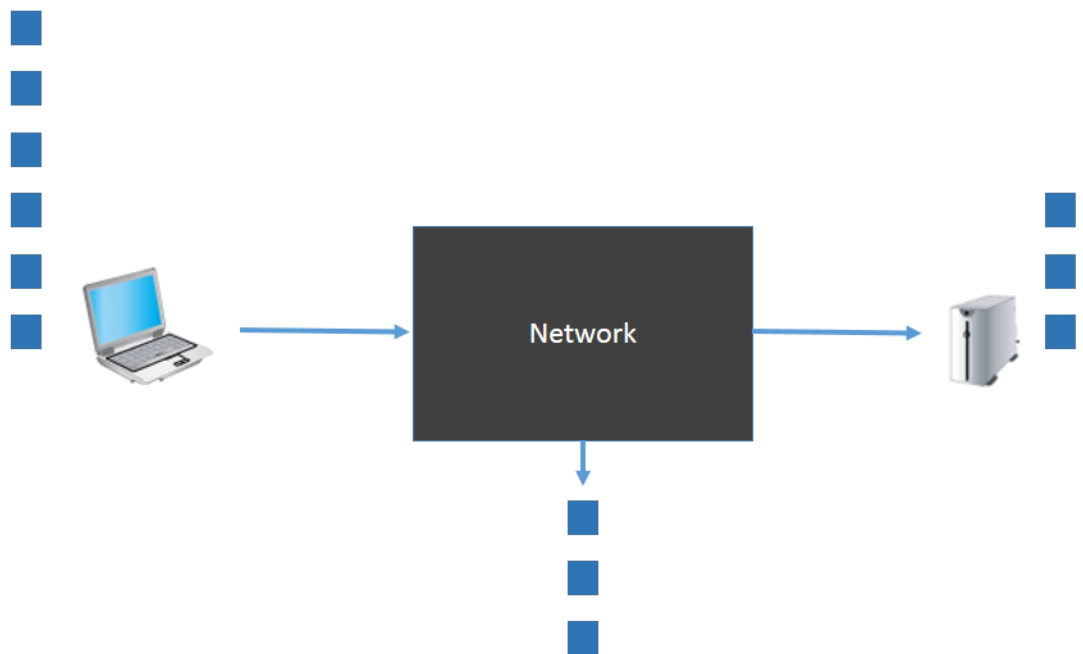


Figure 3: Black Box Principle in the Presence of Congestion

In Figure 3, a TCP sender is injecting data segments into the network, which is from the TCP sender's perspective a "Black Box". The network is facing congestion, and as a consequence, only half of the data segments have reached the TCP receiver,

and the other half has dropped by the network. Acknowledgments of the received segments will be sent to the sender.

3.3.2. Ideal Wireless Segments Loss

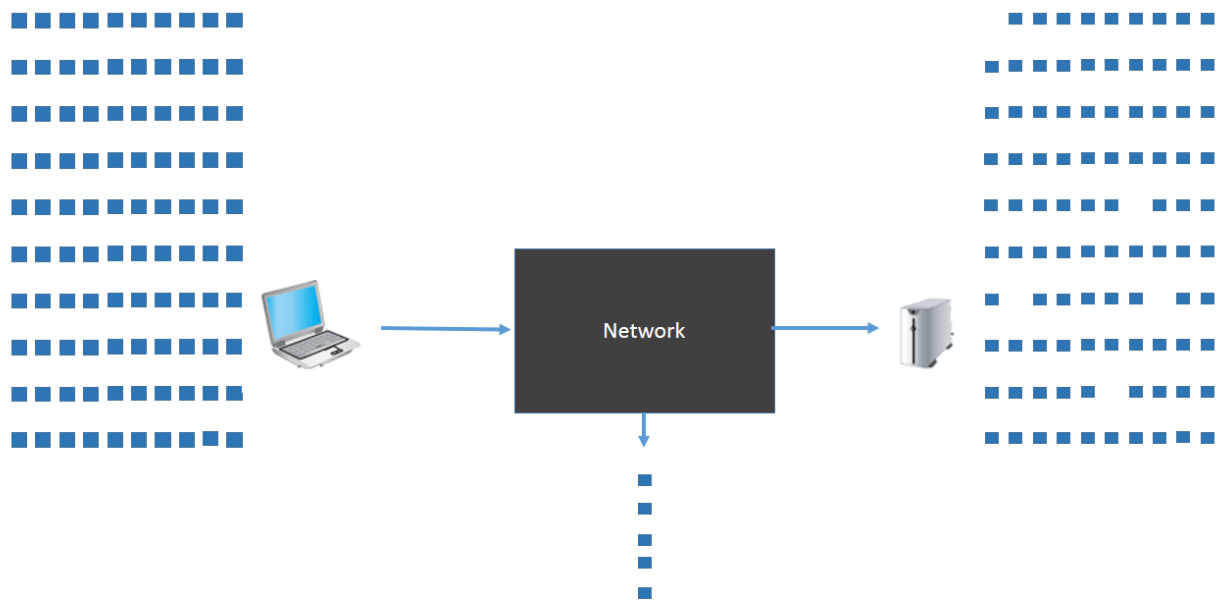


Figure 4: Black Box Principle in the Presence of Random Loss

In this scenario, a TCP sender is sending a stream of data segments into the network, the network has unreliable wireless links which drops data segments. The result will be that most of the data segments will reach the destination node, acknowledgments from the receiver will be sent back to the sender, and very few data segments (in the range of 1-5%) will be dropped.

3.3.3. Actual Scenario

In an actual scenario, the network may have many paths with numerous users sharing the network. Further, the network may have many links with different connections. As an example, a TCP client can be in one country, and the TCP server is in totally different geographic region. As a result, the data segments can face either of the two kinds of data loss.

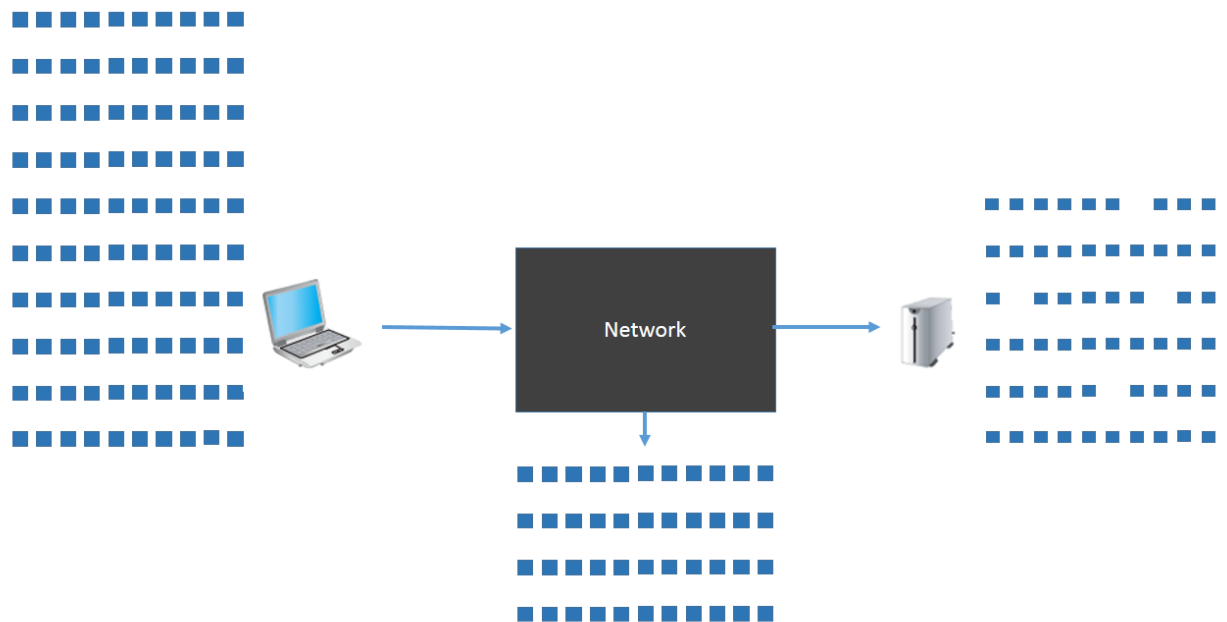


Figure 5: Black Box Principle in the Presence of Congestion and Random Loss

In such a case, the data sent through the network by the TCP sender has gone through hybrid networks, and has experienced loss due to wireless links and congestion. This scenario also signifies the importance of the fundamental principle of TCP, the end-to-end approach to congestion control. In such a scenario, the TCP sender needs to be able

to decide whether to (a) decrease the *cwnd* to the correct network capacity and prevent congestion collapse, or (b) keep the *cwnd* the same, because random loss is not a predictor of future congestion collapse

The key contribution of the FTAT approach is in implicitly determining the cause of data loss, and then adjusting the congestion window in the correct manner. The congestion window is degraded only when necessary – when the loss is actually caused by congestion.

3.4. FTAT Approach

FTAT and Westwood share the same principle of using the feedback of returning acknowledgments to measure the network capacity and adjust the *cwnd* accordingly. However, Westwood's filtering mechanism has its own drawbacks. Filtering the measured capacity samples is good from one point of view: it results in an averaged sample that is not greatly affected by loss. On the other hand, the filtering mechanism assumes that the network has reached its capacity and does not instantly reflect the actual capacity of the network at the time of the sample measurement. This can be shown through simulations conducted using ns-3 implementation and the Linux-stack kernel. This causes Westwood to fall in the same category as NewReno in not predicting the actual network capacity and as a result degrading the throughput.

The mechanism employed by FTAT of testing the network when a loss is detected to implicitly determine the cause of the loss, and accordingly adjusting the *cwnd*, is shown that it greatly predict the cause of the loss, and as a result produce better throughput. To

achieve the “testing”, the same environment when the loss occurred is duplicated, then the delivered data bytes to the receiver is computed, and the *cwnd* is adjusted accordingly. Computing the capacity of the network without filtering the samples is good from the point that it reflects the actual capacity of the network at the time of the sampling and hence can increase the *cwnd* instead of decreasing the *cwnd* as in the congestion case.

The next graph demonstrates the testing mechanism effect on the *cwnd* throughout the connection, in random loss case, and in congestion case.

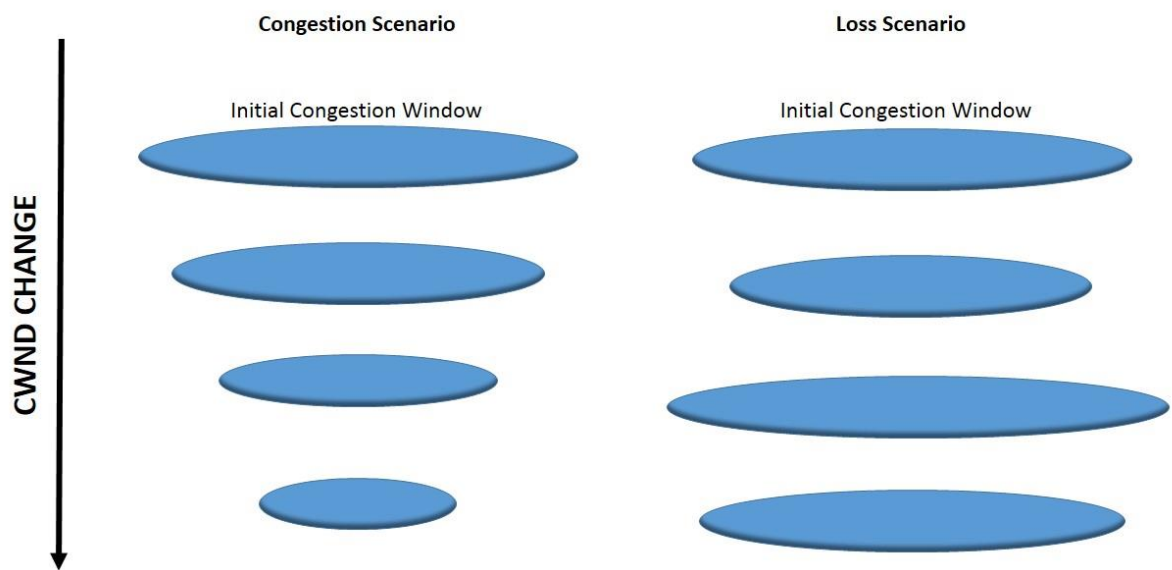


Figure 6: Adaptive Transmission Effect

In the graph it is shown that whenever the testing mechanism is applied, if the cause of the loss was congestion, the result will be always a reduction in the *cwnd* to the correct

capacity of the network. When the loss cause is random loss that has no relevance to the network capacity, the result of the testing is an adjustment of the *cwnd* to the right capacity of the network, which can lead sometimes to reduction in the *cwnd* and other times expansion in the *cwnd*.

3.5. TCP FTAT Congestion Control

TCP FTAT is a new end-to-end congestion control algorithm, which is a modification of Reno that does not require modifications to the TCP structure, and only requires installation in the TCP sender side. TCP FTAT achieves much higher throughput/goodput gain over the other TCP congestion control algorithms. By way of ns-3 simulations, we show that it can achieve more than 22900% and 8500% goodput gain over NewReno and Westwood+, respectively, in congested networks as well as in wired/wireless/hybrid networks due to its adaptive mechanism in adjusting the *cwnd* to the right network capacity and ensuring the delivery of the lost segments in a timely manner. In high bandwidth-delay product networks, TCP FTAT outperforms most of the TCP congestion control algorithms in the throughput/goodput gain. TCP FTAT does not degrade the *cwnd* dramatically each time a loss occurs, as FTAT is sensitive to the nature of the data loss. FTAT does not degrade the *cwnd* directly when a loss occurs, instead the effect of loss is only observed in the overall network bandwidth measurement.

The congestion window, which characterizes the behavior of the congestion control algorithm, has a specific pattern in the case of NewReno and other similar congestion

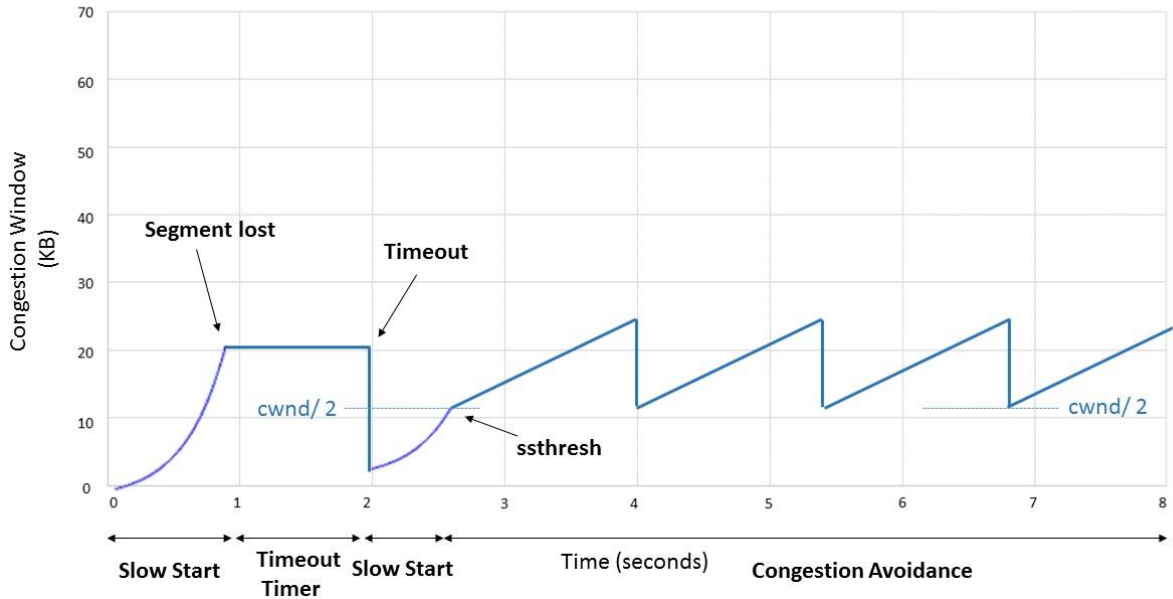


Figure 7: NewReno Congestion Window Pattern

control algorithms, which try to “guess” the available bandwidth in the network. For example due to the halving of $cwnd$ when three duplicate acknowledgments are received, NewReno’s $cwnd$ almost follows a “sawtooth pattern”, the following graph emphasize the sawtooth pattern of NewReno.

TCP FTAT’s $cwnd$, on the other hand, does not follow such a specific pattern; FTAT uses an *Adaptive-Increase Adaptive-Decrease* paradigm, which measures the network’s available bandwidth, and uses these measurements in order to determine the inflation or deflation of $cwnd$.

FTAT is based on two states, *Adaptive Transmission*, and *Additive Increase*. The *Adaptive Transmission* measures the network capacity in a duration of one RTT, then the

data bytes measured is used to adjust the *cwnd*. If the measured capacity indicates that the available bandwidth is lower than the current size of the congestion window, the result is *cwnd* degradation to the correct capacity of the network. If the measured capacity indicates higher available throughput, however, the *cwnd* is *inflated* to the available network capacity. This possible increase in the size of the congestion window is key to the increased performance afforded by TCP FTAT.

In the *Additive Increase* phase, the *cwnd* increases by a combination of the linear and exponential increase of Reno's two phases: Slow Start and Congestion Avoidance.

The purpose of the *Additive Increase* algorithm is as follows:

1. After an initial estimation of the network capacity, Reno assumes that the network is in a stable state and the capacity of the network has been reached.
2. An *Additive increase* paradigm takes over to probe for any additional changes to the network capacity.

3.5.1. Initial Congestion Window

The *quality of service* for the majority of applications on the World Wide Web faces degradation due to the size of the initial congestion window of AIMD protocols. The initial congestion window of Reno is limited to 4380 bytes, which causes the majority of applications on the WWW (that transmit multiples of hundreds of KB) to take longer times in the start-up phase than it would take in normal transmission [12]; in addition to the large products of bandwidth-delay networks that are in increased deployment, which cause the segments to take a long time to travel to the destination and the

acknowledgment to return to the sender. At the same time, if the congestion window is too large in the start-up phase with no knowledge of the condition of a network, that could lead to congestion and may threaten the stability of the Internet. FTAT adjusts the *cwnd* at the start-up phase to 64 KB, which corresponds to the start phase of Congestion Avoidance in Reno algorithm.

3.5.2. FTAT Algorithm

The connection starts with *cwnd* set to 64 KB, which allows a predictable amount of initial throughput. With the first *cwnd* data bytes sent to the network, the acknowledgments are monitored and counted. After the first round-trip time (RTT) has elapsed, the *cwnd* is set to the network's capacity computed as the data bytes acknowledged during the last RTT. The Additive Increase algorithm starts when the *cwnd* value is equal to or greater than the *Congestion Window Threshold* (*cwndthreshold*), which increases the *cwnd* linearly by one MSS every RTT. If the *cwnd* is less than the *cwndthreshold* or if the *cwnd* value is less than 64 KB as Reno, the *cwnd* increases exponentially by one MSS upon receiving each new acknowledgment.

The *cwndthreshold* stores the value of the *cwnd* just before a segment is lost, and hence if the Adaptive Transmission algorithm sets the *cwnd* to a lower value, the Additive Increase algorithm is acknowledged that the capacity of the network is greater and is probing in a fast-paced for the additional bandwidth. When the *cwnd* reaches the value of *cwndthreshold*, the Additive Increase algorithm is alerted that the *cwnd* is in the range of a previous congestion, and hence the rate of *cwnd* increase is slowed down.

The Adaptive Transmission phase starts when there is an alert of change to the network capacity by way of three duplicate acknowledgements. In the Adaptive Transmission state, the packet with the sequence number reported to be lost is retransmitted, and a new bandwidth estimation procedure is initiated. Upon receiving a new acknowledgement or duplicate acknowledgement, a new packet is transmitted in the network. When the number of duplicate acknowledgements reaches three duplicate acknowledgements, the lost packet is retransmitted, and for any additional duplicate acknowledgement a new packet is transmitted through the network. After a period of RTT, the new bandwidth is measured and the Additive Increase state resumes. If the algorithm in the Adaptive Transmission state and a retransmission time-out (RTO) occurs, the Adaptive Transmission state ends and the bandwidth is computed for the data acknowledged in the elapsed period of the RTT. The occurrence of RTO in the Additive Increase state is treated the same way as the NewReno: the *cwnd* is set to one MSS. Figure 8 shows a finite state machine depiction of FTAT.

FTAT

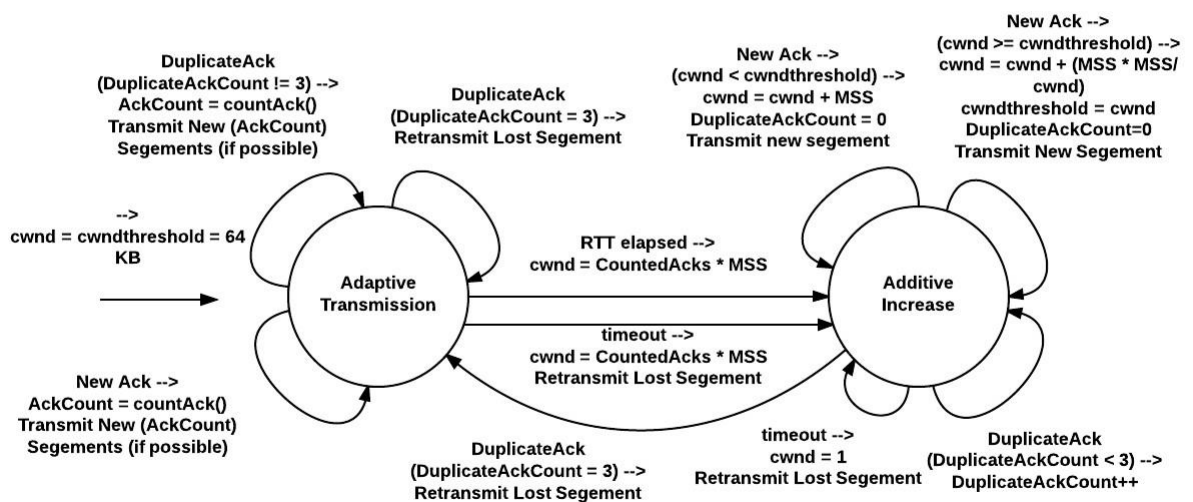


Figure 8: FSM description of FTAT

CHAPTER IV

Implementation of FTAT in ns-3

4.1. The Network Simulator – 3

The *network simulator (ns)* is a discrete-event network simulator for Internet Systems [22, 23, 24]. Network simulators are widely deployed in the networking research community, and ns-2 alone was reported to be used for over 50% of ACM and IEEE simulation-based research papers for the period from 2000 to 2004 [24].

The ns-3 project was adapted by Tom Henderson, Sumit Roy (University of Washington), George Riley (Georgia Tech.), and Sally Floyd (ICIR) to address the weaknesses of ns-2, mainly in aligning with how research is currently conducted, and to improve the credibility of the network simulator. ns-3 is an open source network simulator intended to replace ns-2, although ns-3 is not considered an extension to ns-2 due to a new implementation which replaces the OTcl API with C++ wrapped by Python, and replaces the guts of the simulator completely, and introduces new visualizers.

4.2. The Implementation of TCP in ns-3

TCP in ns-3 is implemented using several classes that provide reliable transport protocol services and communicate with the network layer. The classes that implement the TCP protocol are TcpSocketBase, TcpSocket, TcpHeader, TcpTxBuffer, TcpRxBuffer, TcpL4Protocol, and the different congestion control algorithms

implementations.

- The `TcpSocketBase` class inherits from `TcpSocket`, and provides the interface required for the application layer to the sockets, and is the base for the different TCP congestion control variants.
- The `TcpSocket` class is an abstract class that contains the attributes for required for a TCP socket.
- The `TcpHeader` class contains the implementation for a TCP segment header.
- The `TcpTxBuffer` class provides a buffer service to the application layer, which allows the data to buffer before send out.
- The `TcpRxBuffer` class provides a buffer for the data coming from the network layer before it is passed up to the application layer.
- `TcpL4Protocol` class provides an interface for the network layer to the sockets, and it is responsible for the interactions with the network layer, and it performs the data checksum for the incoming packets.

Ns-3 provides different implementation of the congestion control algorithms, which inherits from the `TcpSocketBase` class. These algorithms are Westwood/ Westwood+, NewReno, Reno, and Tahoe.

4.3. Implementation of FTAT in ns-3

The `tcp-FTAT` class includes the TCP FTAT congestion control implementation. This class inherits from the class `TcpSocketBase` and provides the required functionalities for the TCP FTAT. The main functions are `ReceivedAck()`, `NewAck()`, `DupAck()`, `EstimateRTT()`, `EstimateBW()`, `NewAckProcessing()`, `CountAck()`,

UpdateAckedSegements(), and Retransmit(). The class diagram is shown in Fig. 1.

- receivedAck() is an inherited function from the TcpSocketBase class. It determines if the received acknowledgement is a new acknowledgement or a duplicate acknowledgement. Based on this check, either the newAck() function or the dupAck() is invoked.

The newAck() function is invoked after receiving a new acknowledgement. The way the *cwnd* is handled in newAck() is dependent on the state Adaptive Transmission or Additive Increase. If the algorithm state is in Additive Increase, the *cwnd* is compared to *cwndthreshold* to determine the rate of increase. If the *cwnd* is less than *cwndthreshold*, then the rate of increase would be exponential, one MSS every new acknowledgement. If the *cwnd* is equal to or greater than the *cwndthreshold*, then the *cwnd* is increased at a rate equal to the *Maximum of $((MSS * MSS / cwnd), 1) + cwnd$* , which increases the *cwnd* by approximately one MSS every RTT. It worth noting here that this formula is specified in RFC2581 [25].

If the algorithm state is Adaptive Transmission, the newAck() function will be invoked once a new acknowledgement is received, at this stage the newAck() function will evaluate the variable *m_pktsRound* to determine the number of data packets acknowledged and transmit the same number of packets to the network.

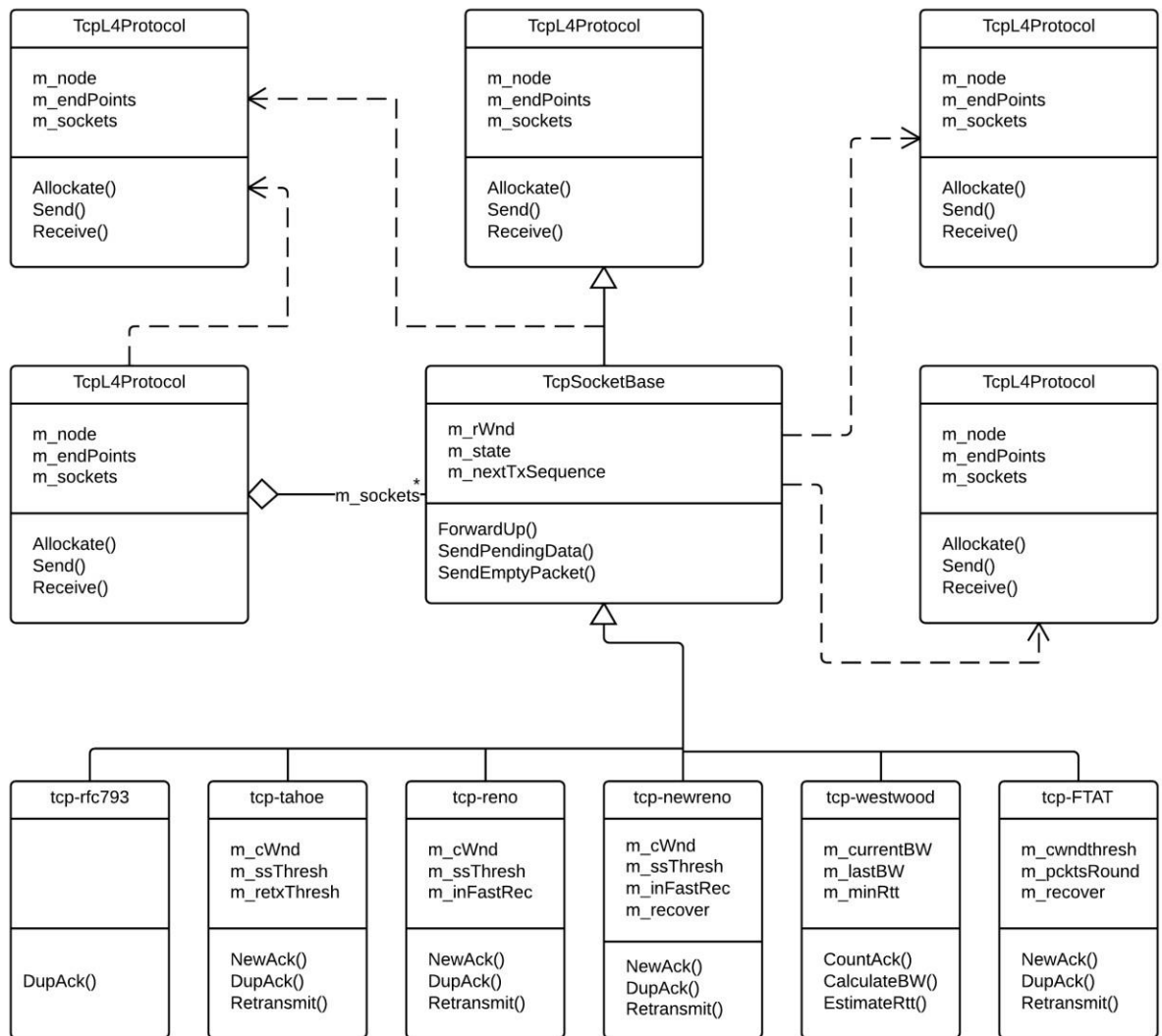


Figure 9: TCP Implementation in ns-3 including FTAT

- The dupAck() function is called after a duplicate acknowledgement is received. If the Algorithm state is Additive Increase, there are no any actions taken by the dupAck() function except when the number of duplicate acknowledgements reaches 3. When the number of duplicate acknowledgements reaches 3, the lost segment is retransmitted and the Adaptive Transmission state is activated without any changes to the *cwnd* or *cwndthreshold* variables. If the state of the algorithm is Adaptive Transmission and the dupAck() is called, on every single call to the

dupAck() function, the value of the `m_pktsRound` is evaluated for the number of data packets acknowledged and a new packet is transmitted to the network.

- The `estimateRTT()` function is called to calculate the last RTT. In `tcp-FTAT`, `estimateRTT()` performs the following two tasks:
 - Perform the last RTT calculation.
 - Schedule a new bandwidth measurement for a period of RTT.
- The `estimateBW()` function is called by `estimateRTT()` after the RTT period has elapsed and it is time to adjust the `cwnd` to the estimated bandwidth and deactivate the Adaptive Transmission. The bandwidth is measured as the maximum of (`m_SegmentSize`, `m_ackedSegements * m_SegmentSize`) where the `m_SegmentSize` is the MSS and `m_ackedSegements` is the number of acknowledged data packets during the state of the Adaptive Transmission and it is reset after setting the `cwnd` to prepare for a new measurement.
- The `newAckProcessing()` function performs the housekeeping for the Adaptive Transmission state. It is called by the `estimateBW()` function and returns the control to `estimateBW()`.
- The `countAck()` and `updateAackedSegements()`, are two functions of the `Westwood+` which perform the counting of the number of data packets acknowledged. It is a novel procedure and gives an accurate calculation for the data packets and takes into account the delayed and accumulative acknowledgements. They are called from the `receivedAck()` function, First the `countAck()` function is called to calculate the number of acknowledged packets, and then the `updateAackedSegements()` is called to update the `m_ackedSegements` variable.

- The Retransmit() function is called after a RTO occurs, it performs the retransmission by calling DoRetransmit() of the TcpSocketBase class, and deactivate the Adaptive Transmission if active and calls the EstimateBW(), or adjusts the *cwnd* to one MSS if the algorithm is in the Additive Increase state.

CHAPTER V

TCP FTAT Linux-Stack Implementation

5.1. Introduction

The Linux operating system has been the most popular choice for many networking applications for more than a decade. These applications include server-side technologies, embedded systems, and a significant number of research efforts in the area of computer networks. Linux also provides the capability of producing customized networking kernels for different networking applications.

Although there are many benefits to using Linux in networking research and applications, it lacks good documentation for its TCP kernel source code, which, in turn, requires significant effort in reading the source code and to get the required information from different resources in order to identify the correct changes to make.

5.2. TCP Congestion Control in Linux

The Linux kernel source code is implemented in C programming language. TCP FTAT is currently implemented in copies of source code for linux-source-3.2.0 and linux-2.6.36. The first attempt to implement TCP FTAT in linux-source-3.2.0 was successful,

and it was recompiled in Ubuntu 12.04 LTS. The second implementation was in linux- 2.6.36, and it was to align with the DCE framework in order to conduct live simulation with the Linux TCP kernel stack.

The TCP protocol implementation in Linux is shown in Figure 10 and Figure 11. Since the congestion control implementation of TCP FTAT is only in the sender-side, the framework has not been changed; only additional congestion control implementation has been added. There are currently 13 congestion control protocols available in Linux, which are Cubic, Reno, BIC, Westwood, Highspeed, Hybla, HTCP, Vegas, VenO, Scalable, LP, Yeah, and Illinois. A brief description on each of these algorithms are as follows.

- TCP Tahoe [1]: The original congestion control by van Jacobson, which consists of Slow Start, Congestion Avoidance, Fast Retransmit.
- TCP Reno [26]: A modification of TCP Tahoe, with the addition of Fast Recovery. This algorithm later became the de facto standard.
- TCP NewReno [2]: A modified version of Reno, also became a standard.
- TCP BIC [27]: Binary Increase Congestion, where the *cwnd* grows more rapidly than NewReno by doing binary search to reach the middle point of the *cwnd* when the congestion was last observed, and then grow rapidly before reaching the congestion point, then slows down the rate of growth when the congestion point is reached, then the window grows rapidly again in search of another congestion point.
- TCP Cubic [28]: The current default congestion control algorithm in Linux, it is designed to address for the high-speed networks, and it is the successor of TCP

BIC.

- TCP Scalable [29]: A congestion control algorithm designed on the idea of making the time of recovery from a congestion constant and unrelated to the congestion window size.
- TCP HighSpeed [30]: A congestion control algorithm that uses a cutoff point to determine the increase factor and the decrease factor in the AIMD paradigm.
- TCP HTCP [31]: The HTCP uses the time since the last congestion as a factor in increasing the congestion window. It has an accurate function based on the RTT to measure the queue size of the bottleneck link along the path, and it uses the measurement to adjust the congestion window decrease factor.
- TCP Vegas [7]: Uses the measurement of the RTT to determine the state of the congestion in the connection and as a result, decreases or increases or maintains the congestion window size.

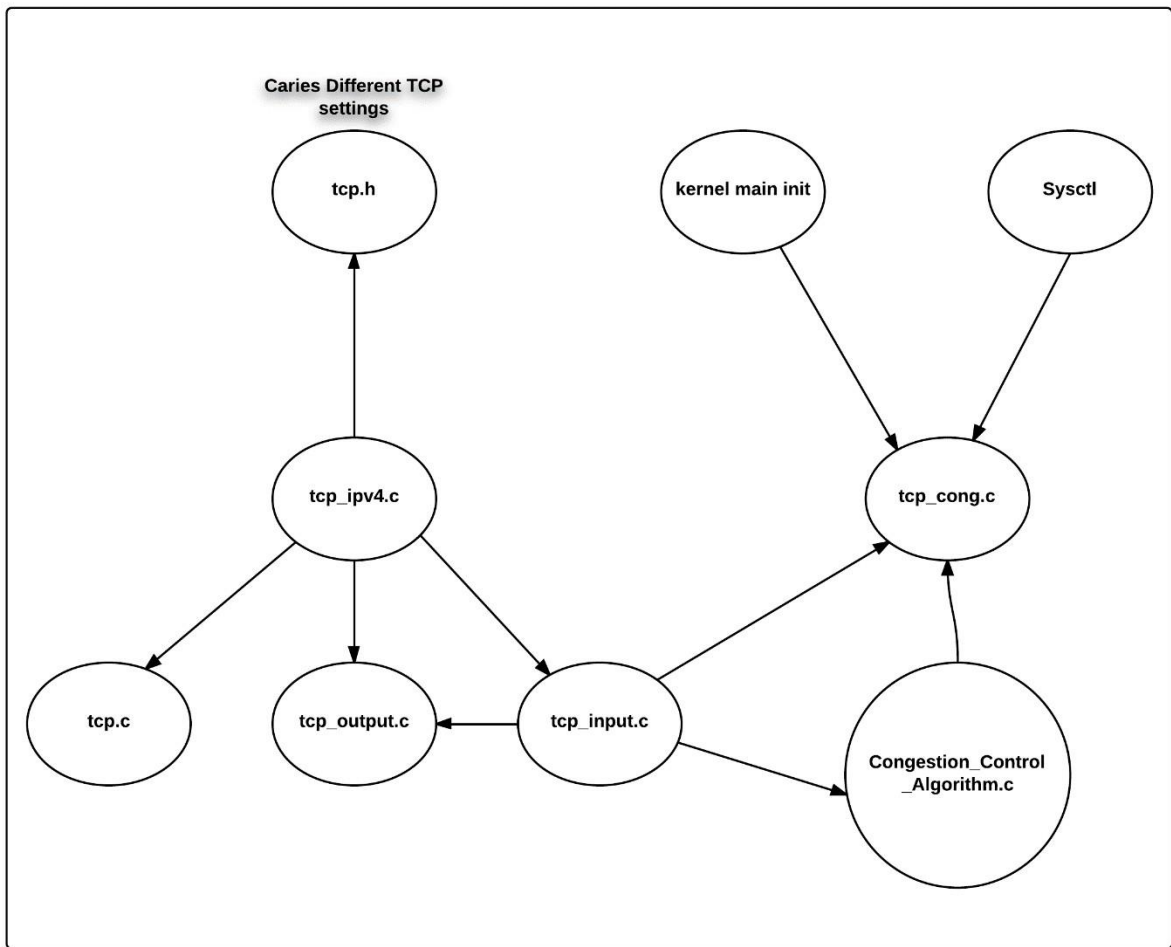


Figure 10: Classes interaction in Linux

- TCP Westwood [5, 6]: An Additive Increase Adaptive Decrease paradigm algorithm that uses the returning acknowledgement as implicit feedback to determine the congestion window size.
- TCP Illinois [32]: Congestion control algorithm that uses the delay of queuing to calculate the factors of the congestion window increase and decrease.

- TCP Hybla [33]: Attempt to determine the increase rate of the congestion window based on the measured RTT to ensure flows fairness. It uses a reference value for the RTT to determine the state of the connection.
- TCP Veno [34]: Uses the same paradigm as NewReno in adjusting the congestion window, but it tries to detect the random losses based on the delay of queuing, and it reduces the congestion window by a factor of 0.20 not the halve as NewReno.
- TCP LP [35]: A Low-Priority Service congestion control that attempt to utilize the unused bandwidth in a TCP flows.
- TCP YeAH [36]: Yet Another Highspeed TCP, is a congestion control algorithm that uses two modes for the congestion window growth, namely, Slow mode and Fast mode. In the slow mode, it implements the Reno rules of growth to the congestion window. While in the Fast mode, it implements the Scalable rules of growth to the congestion window.

The TCP Congestion Control implementation in Linux uses states to differentiate between different congestion states of the connection. It provides more than just the standard states of NewReno, but allows more control such as reversing the *cwnd* decreases. There are two paths for an additive increase state: Slow Path and Fast Path. For the slow path to be active, there must be a duplicate acknowledgement. The fast path takes place when there are no duplicate acknowledgements and the connection is open. The states used by Linux to determine the state of the connection and hence take a proper action by congestion control are Open, Disorder, CWR, Recovery, and Loss.

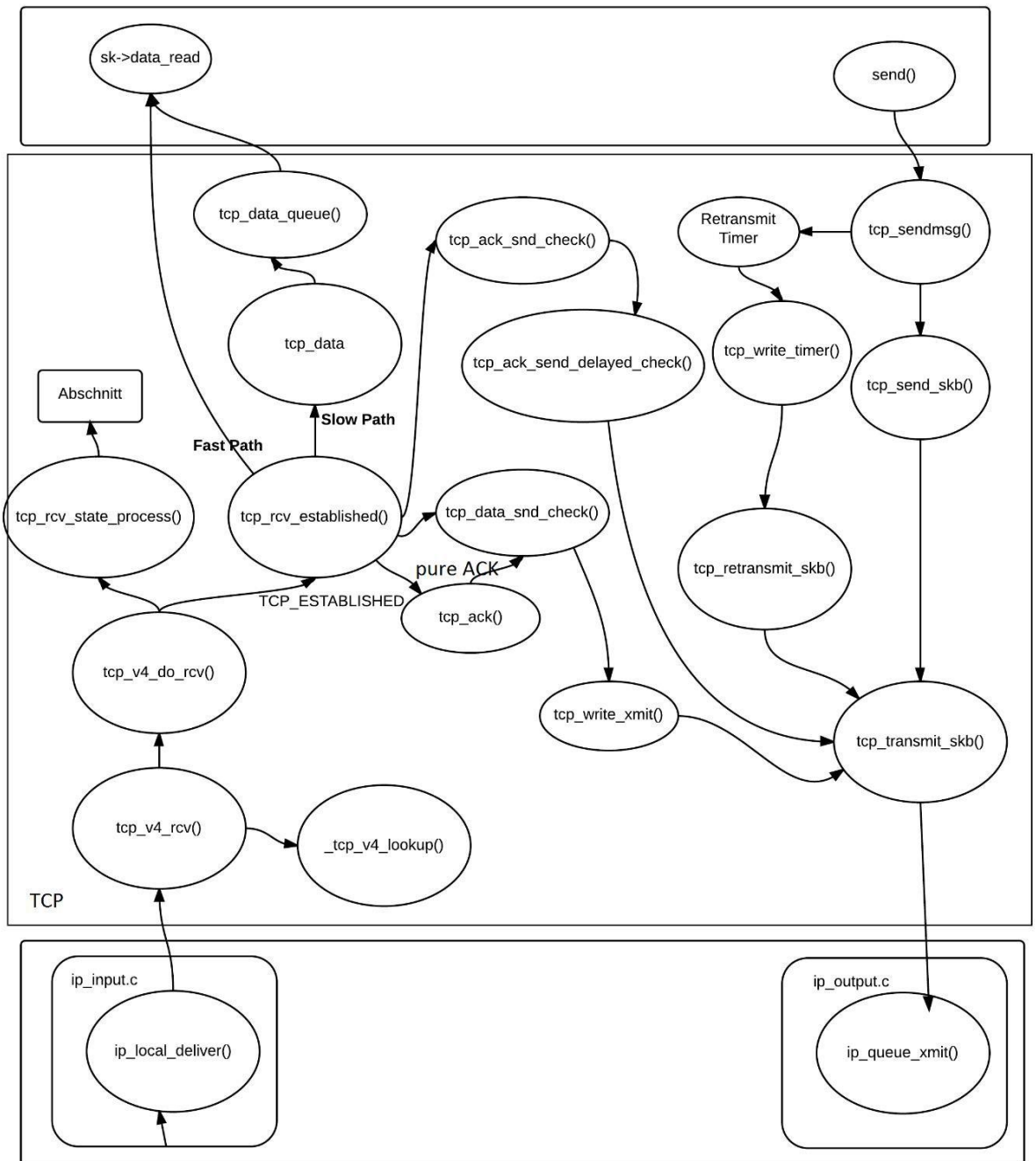


Figure 11: TCP function interaction in Linux

- Open State: when there are no duplicate acknowledgements, the packets received are forward to the Fast Path, and it is the normal state.
- Disorder State: When there are duplicate acknowledgements or SACK, and the packets are forwarded to the Slow Path.

- CWR State: The state, which handles congestion notifications that come from congestion control based on explicit notifications such as ECN.
- Recovery State: It is stated when there is indication of loss and it is time to enter a recovery state, it has the Fast Retransmit operations.
- Loss: It is entered due to loss of RTO or SACK renegeing.

5.3. FTAT Implementation in Linux TCP Stack

The TCP FTAT implementation in the TCP Linux kernel source code resides in the `tcp_FTAT.c` in the `ipv4` sub folder of the `net` folder in the Linux kernel source code.

Additional header file has been added for the `tcp_output.c` in order for FTAT to use some functions in `tcp_output.c` to perform packets transmission in the Adaptive Transmission State.

- `tcp_FTAT_init()`: initializes the variables at the start of the connection.
- `tcp_FTAT_pkts_acked()`: It is called after processing some packets. It adjusts the RTT to the SRTT (Smoothed Round Trip Time) after, and checks the processed packets' RTO status to deactivate the Adaptive Transmission and sets the *cwnd* to the measured capacity of the network, or sets the *cwnd* to one MSS in case of Additive Increase.
- `westwood_acked_count()`: A function from `tcp_westwood`, which performs the counting procedure for the acknowledged bytes after receiving an acknowledgement.
- `tcp_FTAT_Bandwidth()`: called after Adaptive Transmission, performs the capacity calculation of the network, and adjusts the *cwnd*.

- `tcp_FTAT_cwndthreshold()`: returns the value of the `FTAT_cwndthreshold`.
- `tcp_init_data_skb()`: Is called from `tcp_FTAT_probe_skb()`, prepares control bits, and performs the sequence number sliding for the packets which are sent in the Adaptive Transmission state.
- `tcp_FTAT_probe_skb()`: Is called from `tcp_FTAT_probe_skb()` to send new packet in the Adaptive Transmission upon reception of new acknowledgement or duplicate acknowledgement.
- `tcp_FTAT_additive_increase()`: The additive increase function which checks the *cwndthreshold* in comparison to the *cwndthreshold* to determine the rate of increase.
- `tcp_FTAT_adaptive_transmission()`: The center piece function which holds most of the logic of FTAT in TCP Linux. It is called in the Slow path and Fast path, and it performs the following tasks:
 - Checks the RTT to start a new bandwidth measurement.
 - Initializes the `snd_una` variable on the first received acknowledgement to adjust the sequence number of the first unacknowledged byte.
 - Determines if it is time to switch from Adaptive Transmission to Additive Increase and adjust the *cwnd* to new capacity of the network.
 - Determines if it is time to activate the Adaptive Transmission state, if it is not active.
 - In the Adaptive Transmission state, calls `tcp_FTAT_probe_skb()` to send packets.
- `tcp_FTAT_event()`: Switches between states, manages which function gets

called on each state.

- `tcp_congestion_ops()`: Provides the information for the congestion control handler interface.

One important setting for using FTAT is to adjust the sending and receiving buffers to a fair value in order for the end-nodes to be able to buffer a good amount of data in the presence of packets disorder. The current value that has been adjusted for the buffer sizes in Linux and ns-3 is 5 MB.

CHAPTER VI

A Mathematical Model of TCP FTAT

In this chapter we derive a mathematical model of FTAT Adaptive-Increase Adaptive-Decrease mechanism. Because FTAT is a sender side modification of NewReno and for the sake of simplicity, we follow the same arguments developed by Kelly in his paper “Mathematical modeling of the Internet” [14], and that was used in [6] to derive a mathematical model for TCP FTAT.

Theorem: A simplified steady state throughput of the FTAT algorithm is as follow:

Equation 1

$$T_{FTAT} = \lim_{t \rightarrow \infty} x(t) = \frac{B}{2} + \sqrt{\frac{B^2}{4} + \frac{1-p}{RTT^2 \cdot p}}$$

Proof. To develop the model we consider a TCP flow controlled by FTAT, p is the probability of loss signal at the window update, B is the available bandwidth share for the flow, RTT is the mean round trip time. The $cwnd$ is updated upon receiving an Acknowledgement, we assume that the connection is in stable state, and that the $cwnd$ is greater than the $cwndthreshold$, which increase the $cwnd$ by $\frac{1}{cwnd}$ upon receiving new acknowledgement. In the case of congestion signal, the algorithm enters the *Adaptive Transmission* state, and the $cwnd$ is updated after one RTT or time-out by $B \cdot RTT - cwnd$. By the given assumptions, we derive

the following update step for the $cwnd$:

Equation 1

$$\Delta cwnd = \frac{1-p}{cwnd} + (B \cdot RTT - cwnd) \cdot p$$

Since an approximation of the rate at which the $cwnd$ is updated in the Additive increase state is $x = \frac{cwnd}{RTT}$, the time between the update steps in the Additive Increase state or the start of the

Adaptive Transmission state is about $\frac{RTT}{cwnd}$, and the expected change in the rate x per unit time is approximately:

$$\frac{\partial x(t)}{\partial t} = \frac{\left(\frac{1-p}{cwnd} + (B \cdot RTT - cwnd) \cdot p \right) / RTT}{RTT / cwnd} = \frac{1-p}{RTT^2} + \left(\frac{cwnd}{RTT} \cdot B - \left(\frac{cwnd}{RTT} \right)^2 \right) \cdot p$$

Equation 2

$$\frac{\partial x(t)}{\partial t} = \frac{1-p}{RTT^2} + (x(t) \cdot B - x^2(t)) \cdot p$$

Equation 3 is separable differential equation. After separating the variables, Equation 3 can be written:

Equation 3

$$- \partial t \cdot p = \frac{\partial x(t)}{x^2(t) - x(t) \cdot B - \frac{1-p}{RTT^2 \cdot p}}$$

The solution can be obtained by integrating each member

$$\int -p \partial t = \int \frac{1}{x^2(t) - x(t) \cdot B - \frac{1-p}{RTT^2 \cdot p}} \partial x(t)$$

↓

$$x(t) = \frac{x_1 - x_2 \cdot C \cdot e^{-p \cdot t \cdot (x_1 - x_2)}}{1 - C \cdot e^{-p \cdot t \cdot (x_1 - x_2)}}$$

Where C depends on the initial conditions, and the roots of the equation

$$x^2 - x \cdot B - \frac{1-p}{RTT^2 \cdot p} = 0$$

↓

$$x_{1,2} = \frac{B}{2} \pm \sqrt{\frac{B^2}{4} + \frac{1-p}{RTT^2 \cdot p}}$$

And a simplified steady state throughput of the FTAT algorithm can be described as:

$$T_{FTAT} = \lim_{t \rightarrow \infty} x(t) = \frac{B}{2} + \sqrt{\frac{B^2}{4} + \frac{1-p}{RTT^2 \cdot p}}$$

By deriving the steady state throughput of the FTAT algorithm, we are able to show the following corollary.

Corollary: The FTAT congestion control is stable ($T_{FTAT} \leq B$)

Proof. From Equation 4, we can argue that T_{FTAT} is always less than or equal to the available bandwidth, B . To show that, we use the same contradictions in [6], that is if we assume that $T_{FTAT} > B$, this assumption leads to congestion collapse, and this leads to drop probability, p up to 1. As a result Equation 4. will result $T_{FTAT} = B$, this result will contradict the assumption. And by this we can conclude that T_{FTAT} is always less than or equal to B , in other words $T_{FTAT} \leq B$.

CHAPTER VII

Evaluation based comparison of TCP-NewReno, TCP-Westwood+, and TCP-FTAT using ns-3

In this chapter we compare the performance of NewReno, Westwood+, and FTAT congestion control algorithms using ns-3 in order to evaluate the behavior of each algorithm in different networks. NewReno and Westwood+ are the native implementations in ns-3. For wireless connections, a packet error model is installed on the links, and the error rate is denoted as p . For all topologies, unless otherwise stated, the default configuration of the buffer size and MSS are 5MB and 536 bytes, respectively.

7.1. Topology One

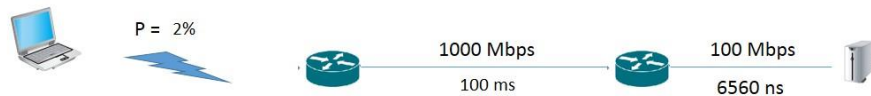
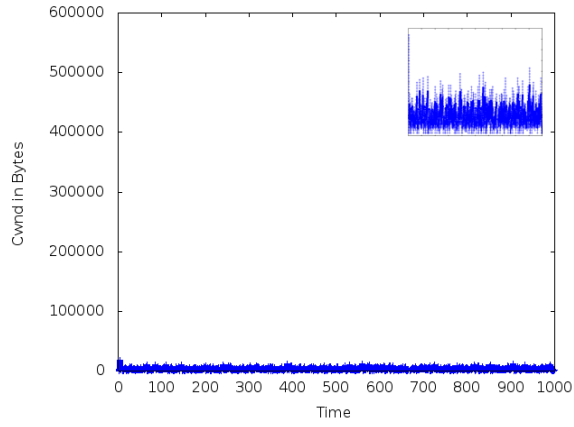
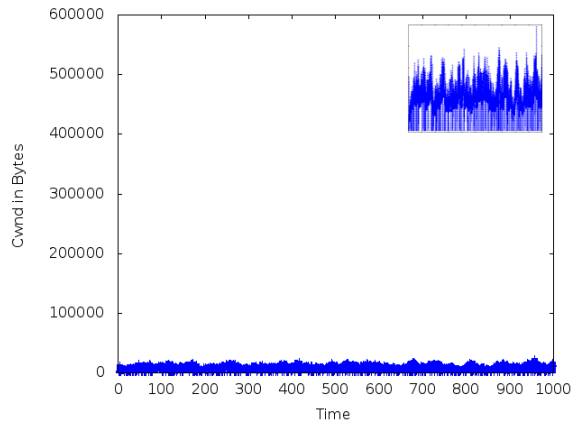


Figure 12: First Topology

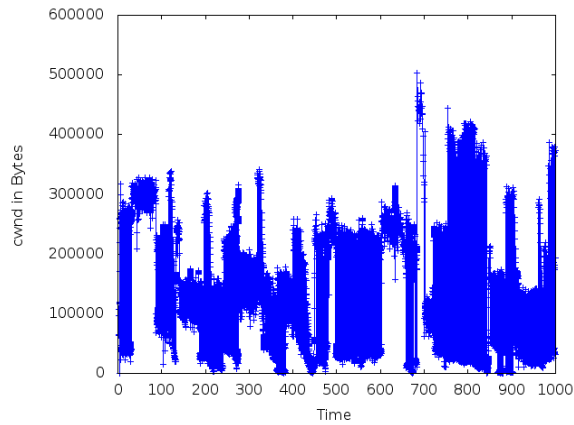
This topology examines the congestion control algorithm when the segments must travel through different connections. The first connection is a wireless connection with 2% packet loss error, the next connection is high Bandwidth-delay product (BDP) network. The last connection is an Ethernet connection. Packets may experience losses due to the wireless connection, and experience large RTT. The simulation time is set to 1000 seconds.



a) **NewReno cwnd**



b) **Westwood+ cwnd**

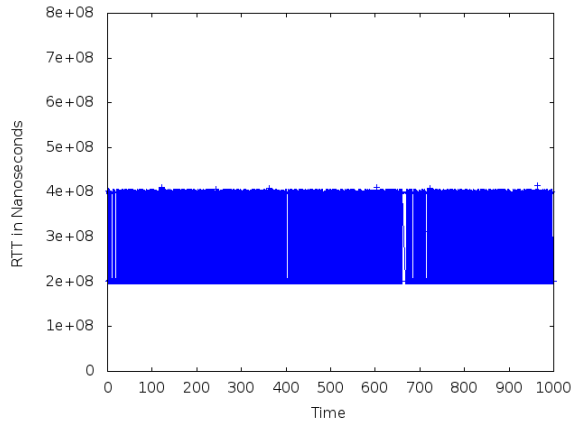


c) **FTAT cwnd**

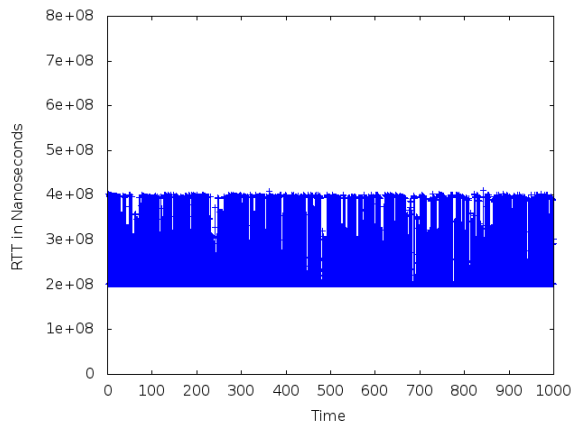
Figure 13: cwnd graphs for Topology one

The congestion window vs time graph (Figure 13) is helpful in confirming the congestion control behavior in different situations, for example NewReno reacts to a retransmission timeout (RTO) by resetting the congestion window to one packet, while in the case of triple duplicate acknowledgement, NewReno halves the congestion window. Westwood+ reduces the congestion Window upon receiving three duplicate acknowledgements, by adjusting it to the last bandwidth measurement obtained, and FTAT starts new bandwidth measurement and enters the Adaptive Transmission which can identify a false alert of congestion and in that case, the congestion window increases. In case of congestion, FTAT reduces the congestion window to the available network bandwidth.

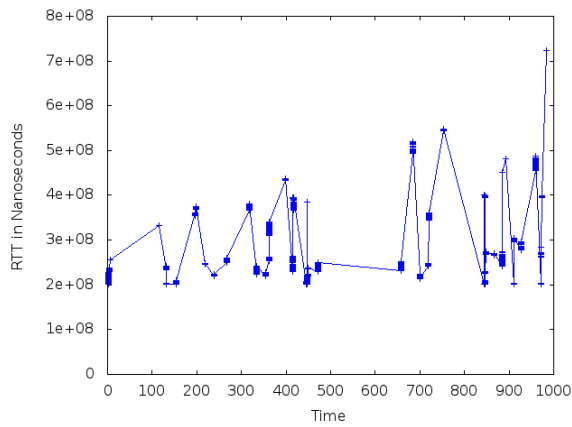
The *cwnd* graph of NewReno shows the behavior of the slow start and the congestion avoidance. Because of the long RTT and the loss rate, the window is going to one MSS more often, and it does not grow more than 12,000 bytes. On the other hand, Westwood shows more growth to the *cwnd*, also goes to one MSS more often, and the window does not grow more than 20,000 bytes. The *cwnd* of FTAT shows more growth even under the long RTT and the loss rate, the *cwnd* growth up to 500,000 Bytes, and goes to one MSS less often.



a) **NewReno**



b) **Westwood+**



c) **FTAT**

Figure 14: RTT graphs for topology two

The three graphs (Figure 14), show almost the same range. NewReno shows almost constant behavior along the connection time. Westwood+ exhibits slight difference in the RTT at some points. While FTAT shows totally new behavior for the RTT, which gives sometimes shorter RTTs and other times a bit longer RTTs. This is due to the nature of the FTAT algorithm, which tries to adapt to the network available bandwidth with time, so the algorithm tries to send more data in the network, but because this is a heterogeneous network with different links and different propagation delays with packet loss probability of 2%, data gets lost and timeout occurs, which increases the RTT.

The graphs in Figure 15 shows the sequence number advancement for the packets throughout the connection life time.

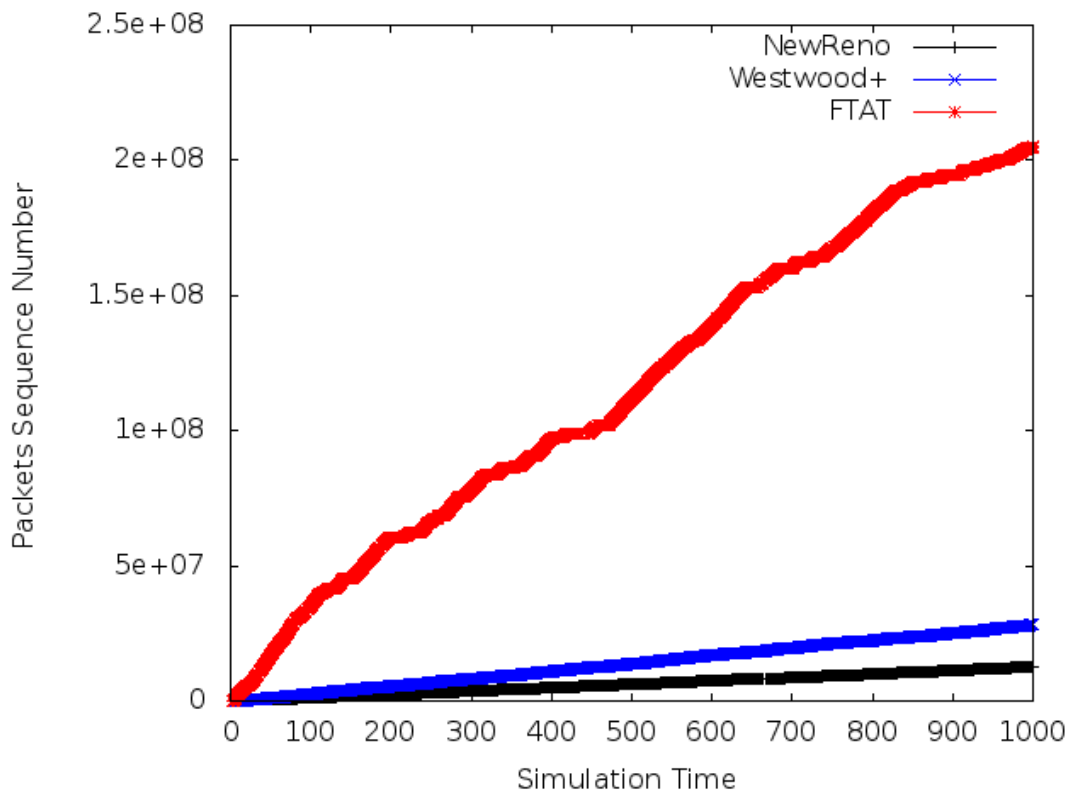


Figure 15: Sequence number topology one

The FTAT algorithm is shown in the red line, while Westwood in the blue line, and NewReno in the black line. The graph shows that NewReno and Westwood+ almost have linear packet sequence number advancement. NewReno reached sequence number 12638881, Westwood+ reached 28204321, and FTAT reached 205139529.

The total throughput for the simulation for the three algorithms is shown in Figure 16. Because throughput might sometimes be deceptive as a measure, goodput is shown for the three algorithms as well. Goodput is calculated as

$$\frac{\textit{Sent bytes} - \textit{Retransmitted bytes}}{\textit{Transfer time}}$$

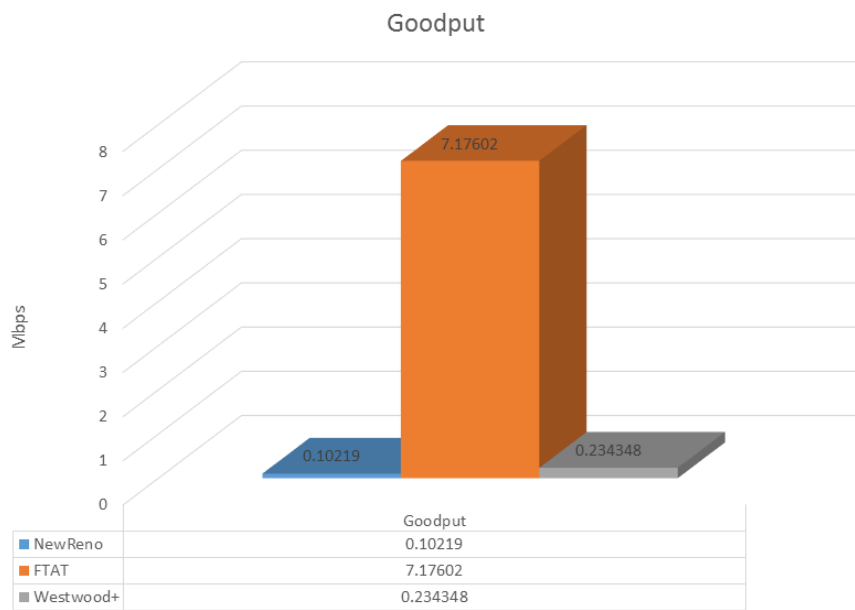
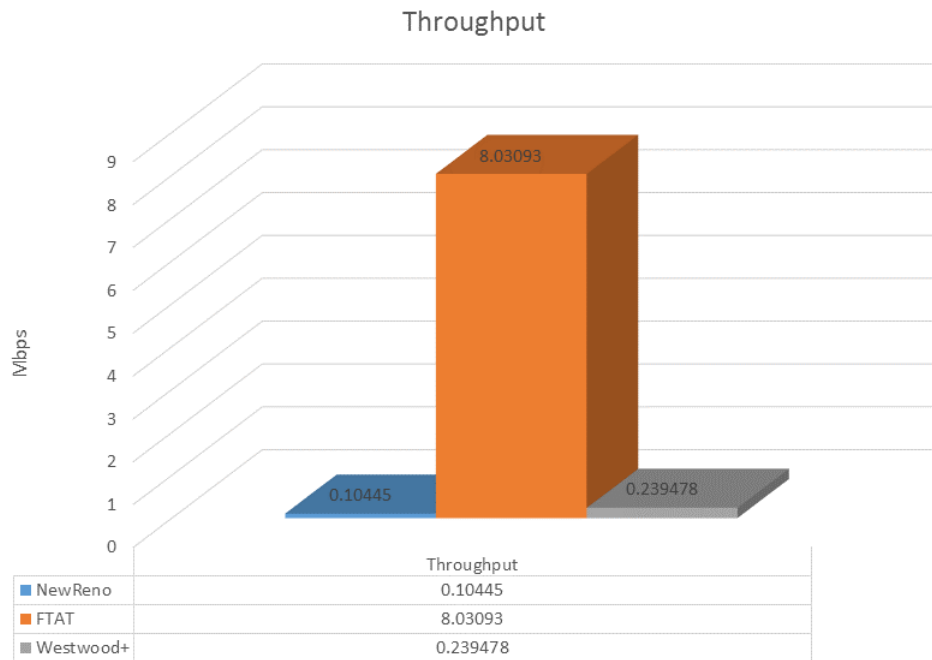


Figure 16: Throughput and Goodput topology one

7.2. Topology Two

In this topology, a network with various traffic directions share a link of 100 Mbps and 10 ms delay with the TCP-variant on the server side, and the receiving node has a packet loss model of 2%. 10 TCP NewReno senders are injecting traffic in the same direction as the TCP-variant, and 10 TCP NewReno nodes are injecting traffic in the reverse direction. The access links are 100 Mbps and 1ms delay time. The simulation time is 100 seconds.

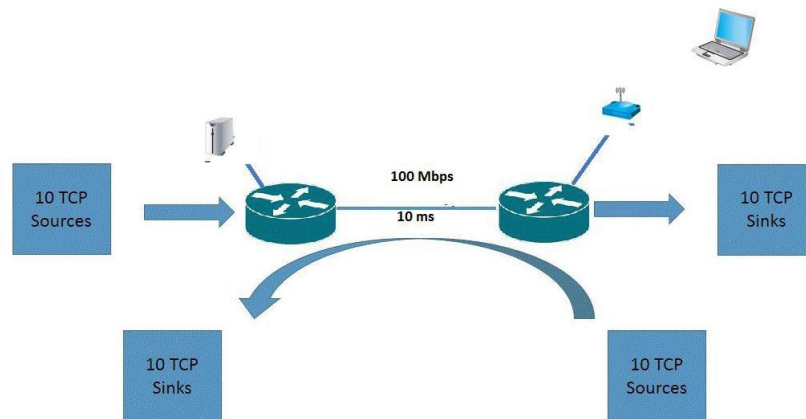
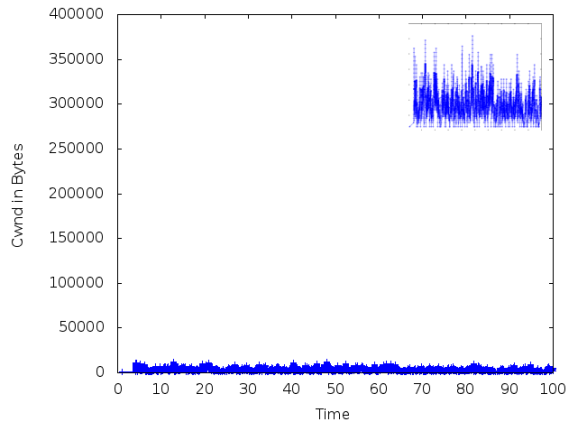
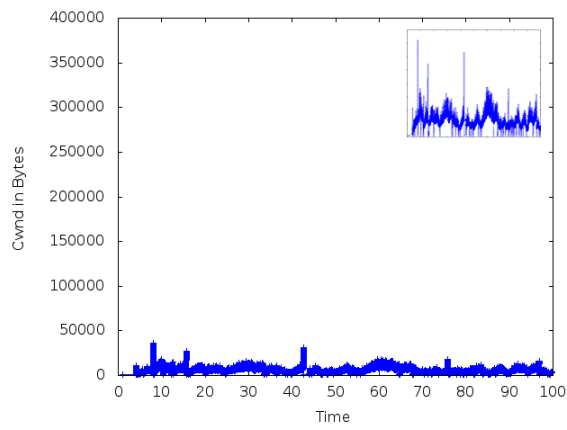


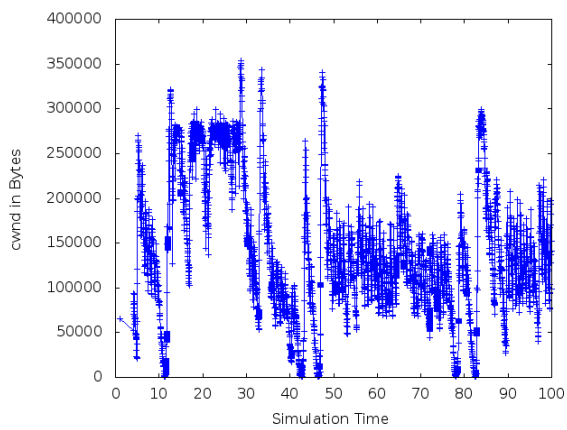
Figure 17: Topology two



a) **NewReno**



b) **Westwood**



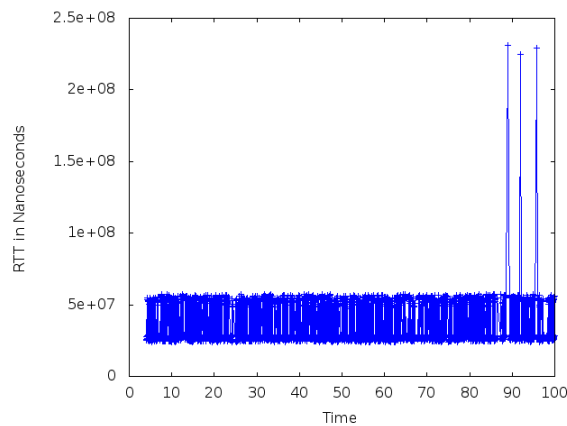
c) **FTAT**

Figure 18: cwnd graphs topology two

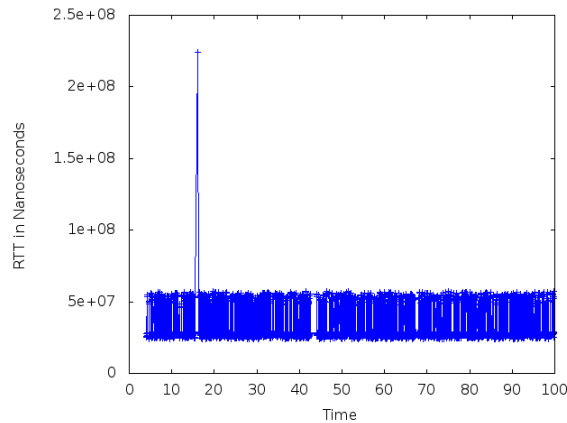
The *cwnd* graphs in Figure 18 show that NewReno *cwnd* growth in the presence of reverse traffic and packet loss rate of 2% did not exceed 12,000 bytes except at the second 50, which was just a notch, and the window goes to one MSS rapidly.

Westwood's *cwnd* shows less fluctuations with a window growth reaching 20,000 Bytes.

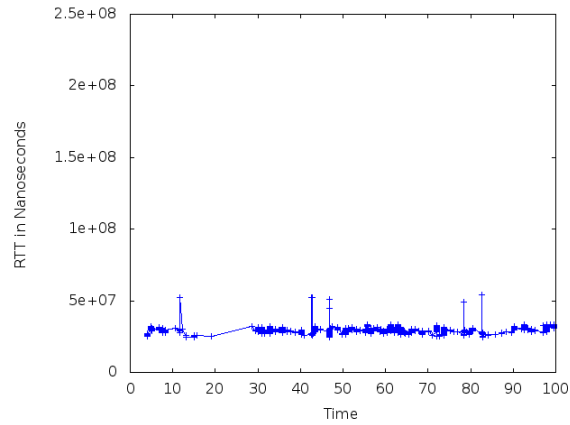
FTAT shows more window growth reached 350,000 Bytes, and goes to one MSS much less often.



a) **NewReno**



b) **Westwood**



c) FTAT

Figure 19: RTT graphs for topology two

The RTT graphs in Figure 19 show that NewReno has almost constant range of fluctuations for the RTT ranging from 25 ms to about 54 ms. Westwood+ has a higher upper limit for the RTT reaches about 60 ms. FTAT has a much lower rate of fluctuations for the RTT, and has a lower average RTT of about 29 ms.

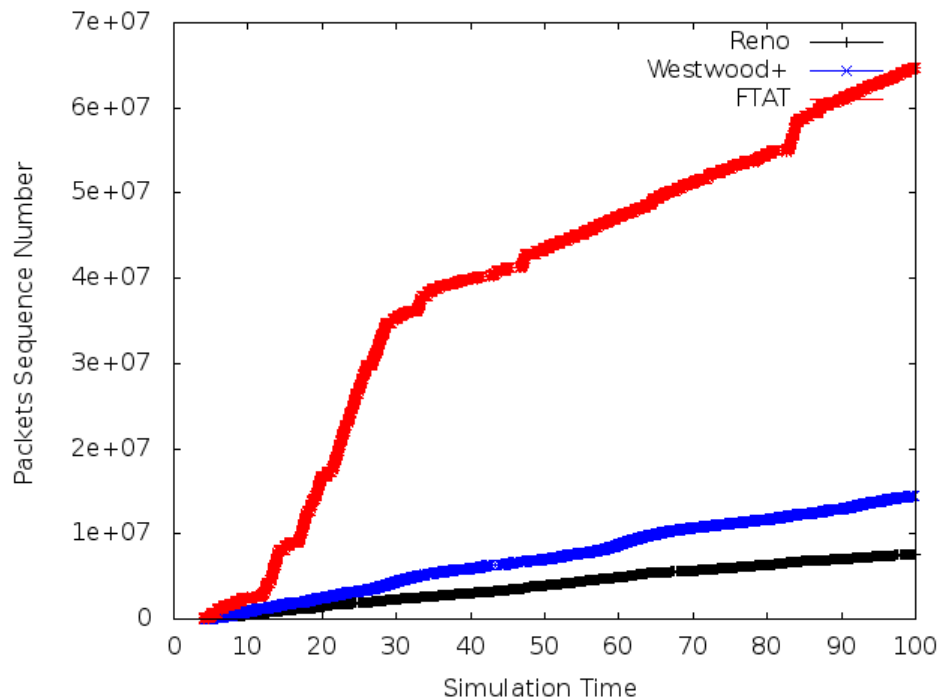


Figure 20: Sequence number topology two

The packets advancement sequence number graph in Figure 20 shows that NewReno

reached 7,661,585 packets transmitted, Westwood+ reached 14,429,121 packets, and FTAT reached 64,793,825 packets.

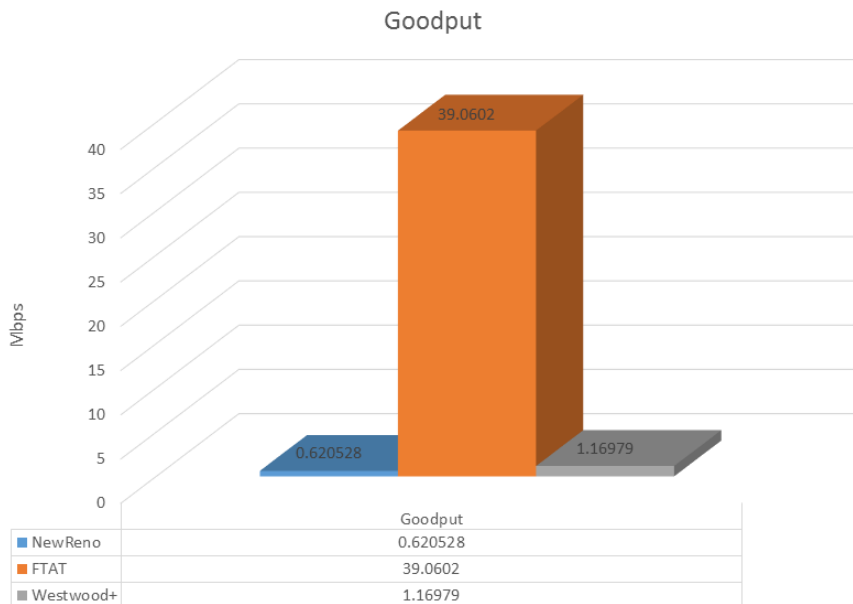
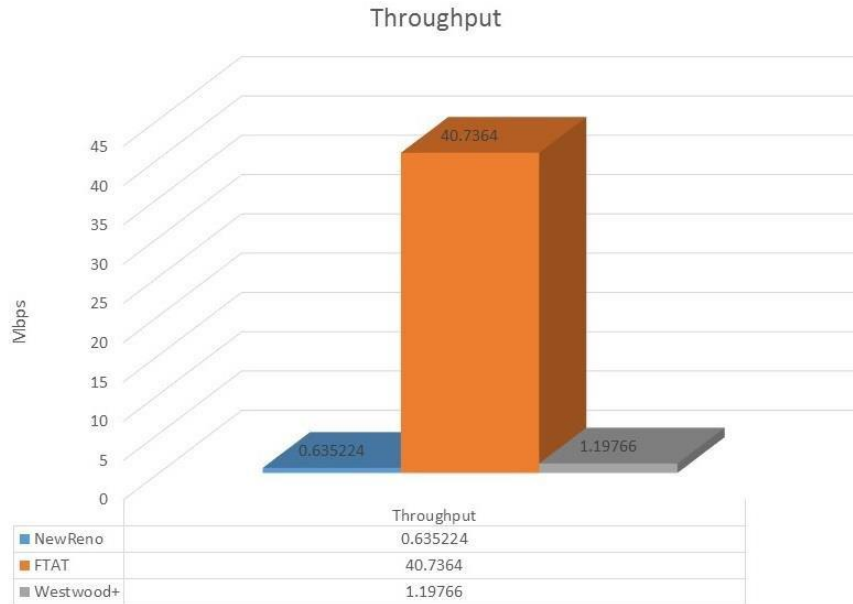


Figure 21: Throughput and Goodput for topology two

In Figure 21, it is shown the throughput/ goodput of the three scenarios. In NewReno scenario, all forward and reverse traffic is controlled by NewReno congestion control, the NewReno flow under test was unable to grab its share of the network available bandwidth of the 100 Mbps, the throughput is measured to be about 0.635 Mbps.

In Westwood scenario, all forward and reverse traffic is controlled by NewReno except the sender of the Westwood flow, which is controlled by Westwood+ congestion control. The throughput graph shows improvement over NewReno, and the measured throughput is computed to be 1.198 Mbps.

In FTAT scenario, again all the traffic is controlled by NewReno except the FTAT sender, which is controlled by FTAT congestion control. Because the high volume of traffic on the bottleneck link, which is 21 TCP flows with 42 nodes, and in different directions. NewReno flows was unable to grab a proper share of the network available bandwidth, and as a consequence, huge bandwidth was left unutilized, and FTAT flow was able to achieve 40.736 alone.

7.3. Topology Three

In this topology, a TCP-variant source is sending persistent traffic to a TCP NewReno sink, and 10 On-Off TCP NewReno sources send traffic in the reverse direction of the TCP-variant. The 10 TCP NewReno sources are on during the periods [250, 500] and [750, 1000], and off during the periods [0, 250] and [500, 750]. The bottleneck link, which determines the capacity of the network, is 2 Mbps and a delay of 1ms. This topology examine the reaction of the congestion control algorithm when the traffic is dominated by reverse traffic, and is used in [15].

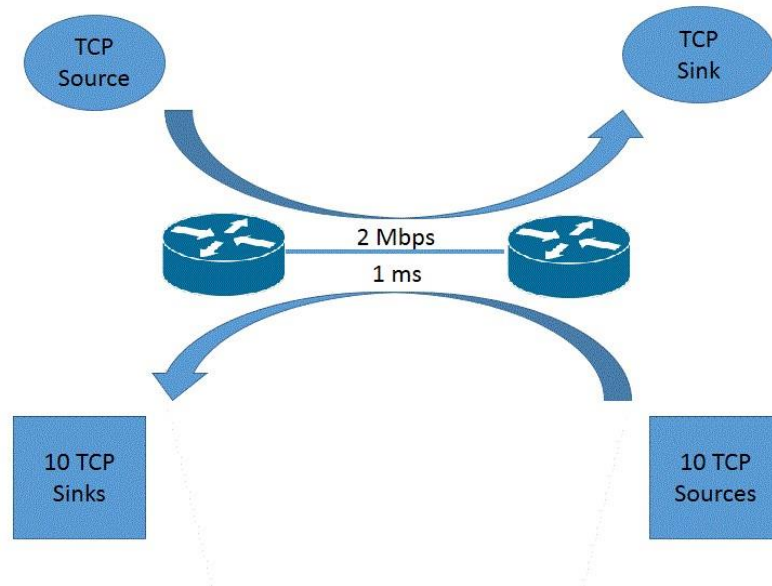
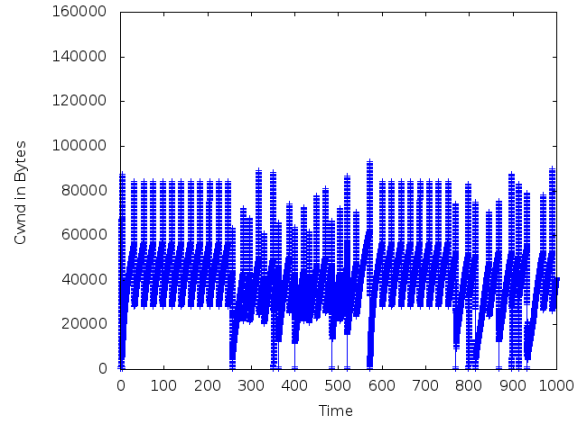
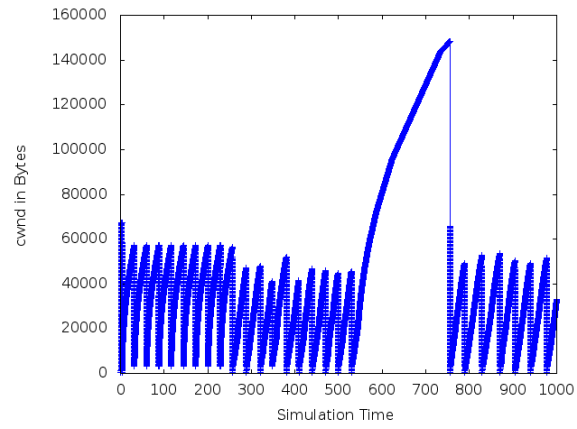


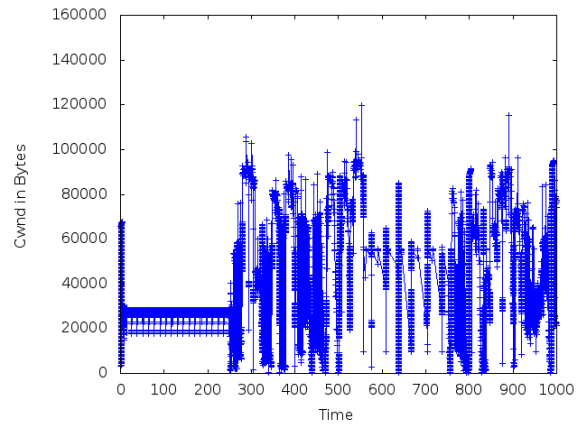
Figure 22: Third Topology



a) **NewReno**



b) **Westwood**



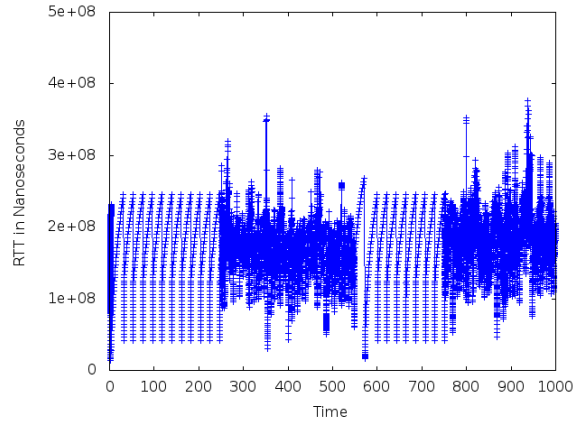
c) **FTAT**

Figure 23: cwnd topology three

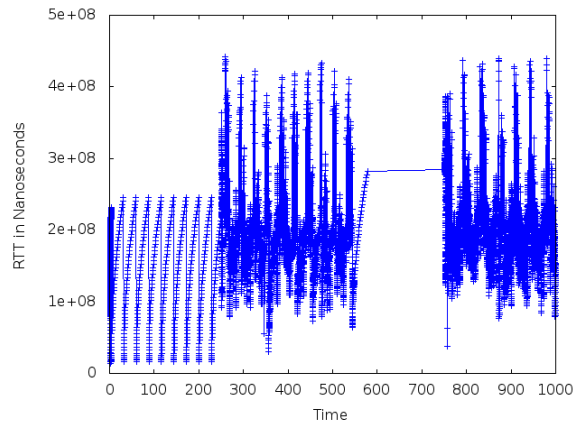
The *cwnd* graph of NewReno shows smooth fluctuations in the period from [0, 250] and [500, 750], and a random fluctuations in the periods [250, 500] and [750, 1000].

Westwood+ *cwnd* graph shows that before the second 250, before the reverse traffic starts, the *cwnd* has behavior similar to that of NewReno, which can be described as smooth fluctuation. After the reverse traffic starts at second 250, there is a lower bound for the fluctuations, at the second 500, after the reverse traffic has stopped, the window increases rapidly until the reverse traffic starts again at second 750.

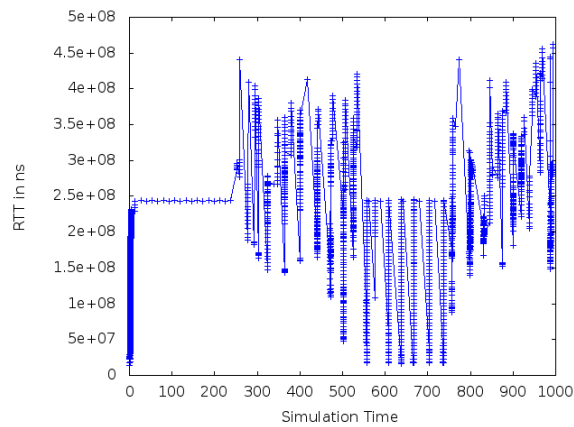
FTAT enters the Adaptive Transmission state when there are loss events, in order to determine the cause of the loss, and as a result the actual bandwidth of the network. The *cwnd* graph shows light fluctuation from the period of [0, 250], where there are no heavy loss events, and as result FTAT behaves much similar as NewReno. in the period [250, 500], the *cwnd* starts to fluctuate heavily, due to the high loss events of the reverse traffic. Although in this scenario this behavior did not reveal more bandwidth, it is very important in high bandwidth-delay product networks such as Geo Sattellite stations, where there are both long delays and high data loss probability.



a) **NewReno**



b) **Westwood**



c) **FTAT**

Figure 24: RTT graphs topology three

The RTT graphs in Figure 24 for NewReno and Westwood shows almost similar behavior for the RTT variations. FTAT shows an almost constant RTT in the

period [0, 250].

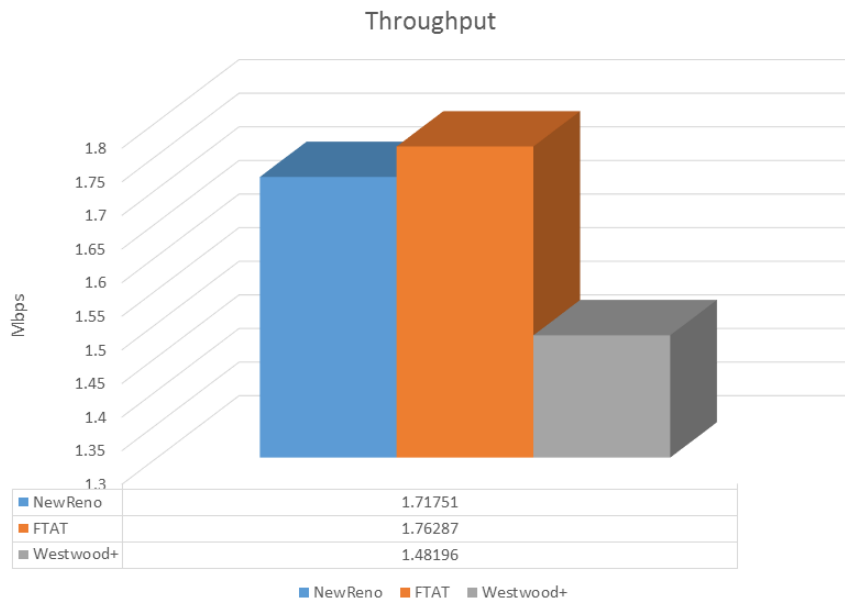


Figure 25: Throughput topology three

Figure 25 shows the throughput of the three protocols. In case of NewReno, the algorithm was very stable throughout the connection, and the graphs shows it has the least fluctuations when the reverse traffic was introduced. This is because the network is limited to the bandwidth of the bottleneck link, which is 2 Mbps. In Westwood scenario, the graphs shows that the Westwood algorithm was affected greatly by the reverse traffic, and there are many occurs of the timeout, and as a result the throughput was affected and reduced. In the FTAT scenario, the graphs shows that the algorithm reacted to the reverse traffic by entering the Adaptive Transmission state frequently, and it achieved slight improvement over NewReno in the throughput for the duration of the connection.

7.4.Fourth Topology: Two-way Geo Satellite Scenario

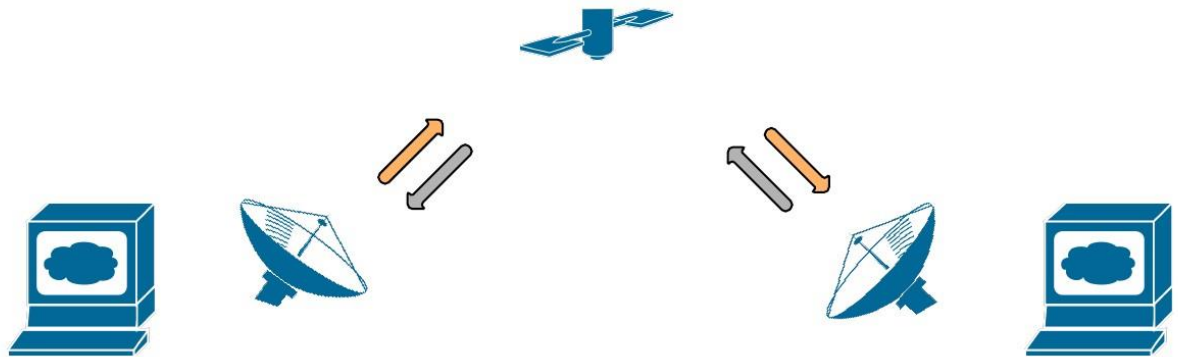
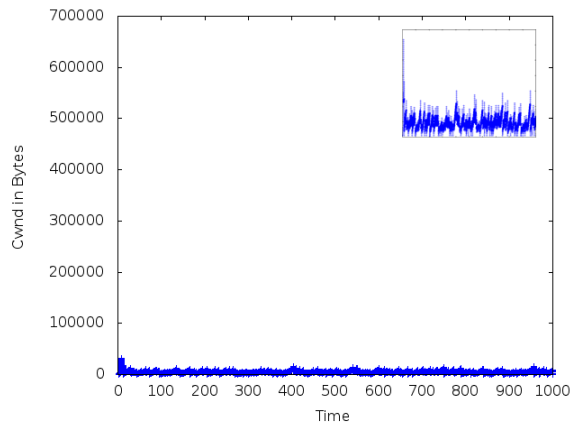


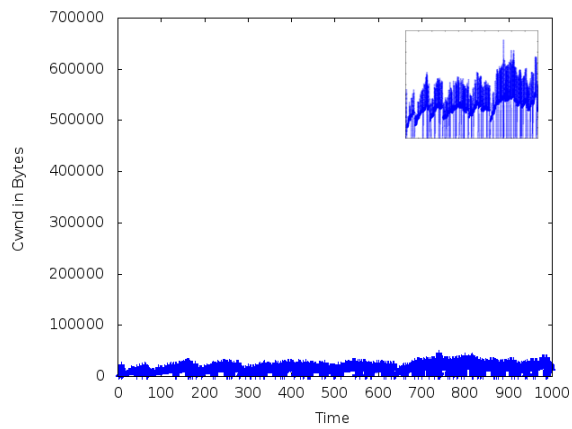
Figure 26: Topology four

In Figure 26, the node on the left side uses the TCP-variants, and the node on the right side uses TCP NewReno. The nodes are sending in both directions on a connection which encounters a packet loss rate of 1% with a long delay of 270 ms. The bandwidth of the connection is 10 Mbps, the simulation time is 1000 seconds, and the packet size is 536 bytes.

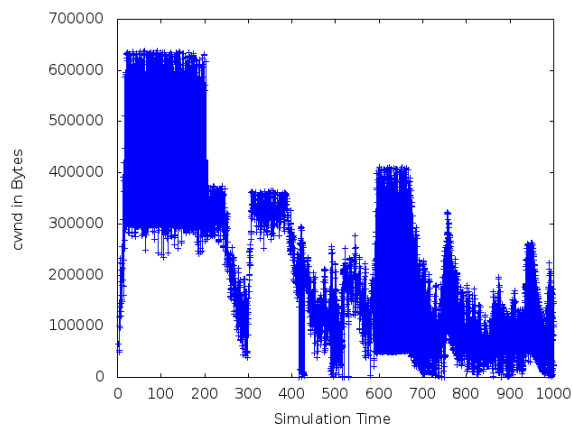
In the first scenario, the left-side node operates with TCP NewReno and the right-side node operates with TCP NewReno. In second Scenario, the left-side node is TCP-Westwood+ and the right-side node is TCP NewReno. And in the third scenario, the left-side node is TCP-FTAT, and the right-side node is TCP-NewReno. The focus of the analysis is on the TCP-variants on the left-side node.



a) **NewReno**



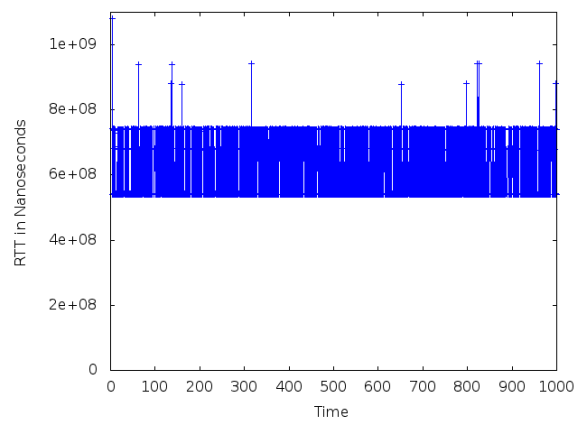
b) **Westwood**



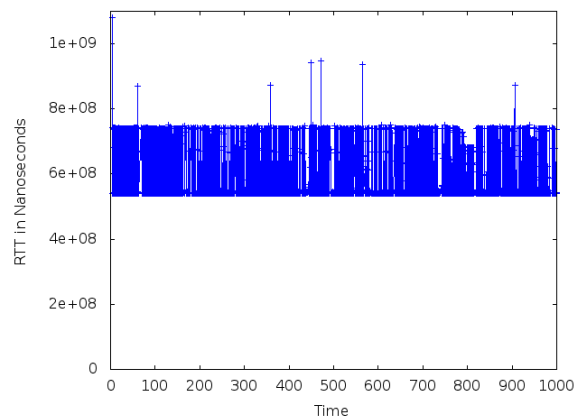
c) **FTAT**

Figure 27: cwnd topology four.

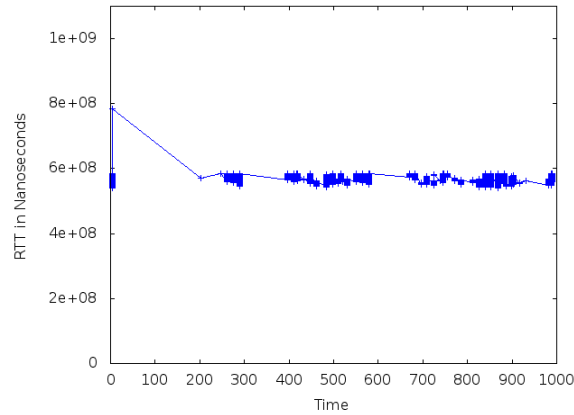
The *cwnd* graphs in Figure 27 show that NewReno failed to reach *cwnd* size of 10,000 bytes, while Westwood reached about 35,000 byte window size, and FTAT reached (in the first 200 seconds) more than 600,000 bytes and then started to fluctuate heavily during the connection due to the long delay and the packet loss probability of 1% in the presence of another connection trying to achieve a bandwidth share on the same link.



a) **NewReno**



b) **Westwood**



c) **FTAT**

Figure 28: RTT topology four

The RTT graphs in Figure 28 shows large values for the RTT, which can be explained by the long delay for a geostationary satellite of 270 ms. NewReno and Westwood have almost identical behavior for the RTT, and the RTT fluctuates between about 54 ms and about 74 ms. The RTT graph for FTAT shows much less fluctuations in the RTT with average RTT of about 57 ms. This is due to FTAT algorithm dropped to one segment much less frequent than NewReno or Westwood, and as a result achieved a state of stability in the presence of long delay, packets drop probability of 1%, and the reverse traffic introduced by the competing connection.

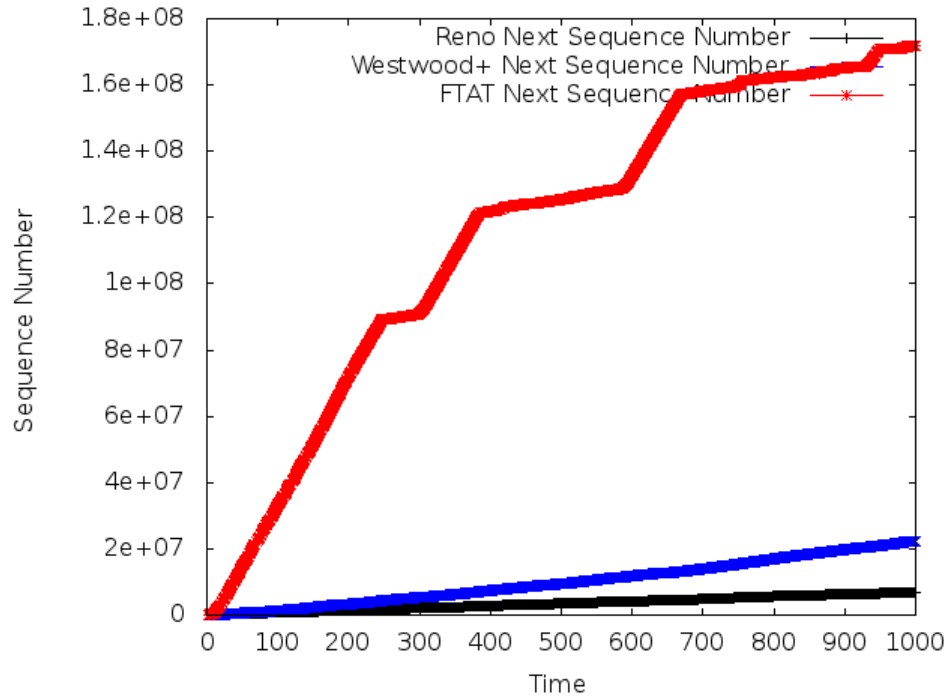


Figure 29: Sequence number topology four

The sequence number graphs in Figure 29 show that during the time of the simulation (1000 seconds), NewReno represented by the black line reached 6,995,873 packet sequence number, while Westwood+ represented by the blue line reached 22,329,761 packet sequence number, and FTAT in red, advanced to reach 171,932,721 packet sequence number.

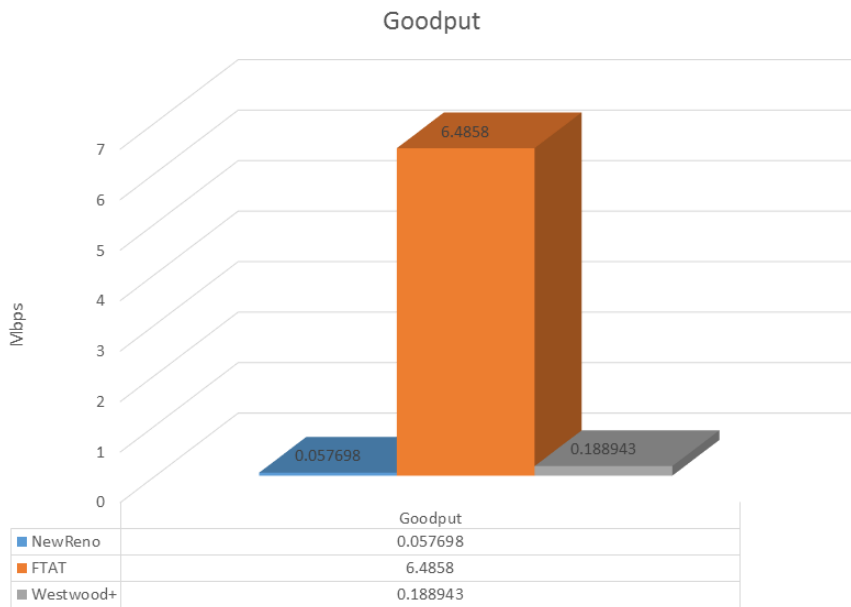
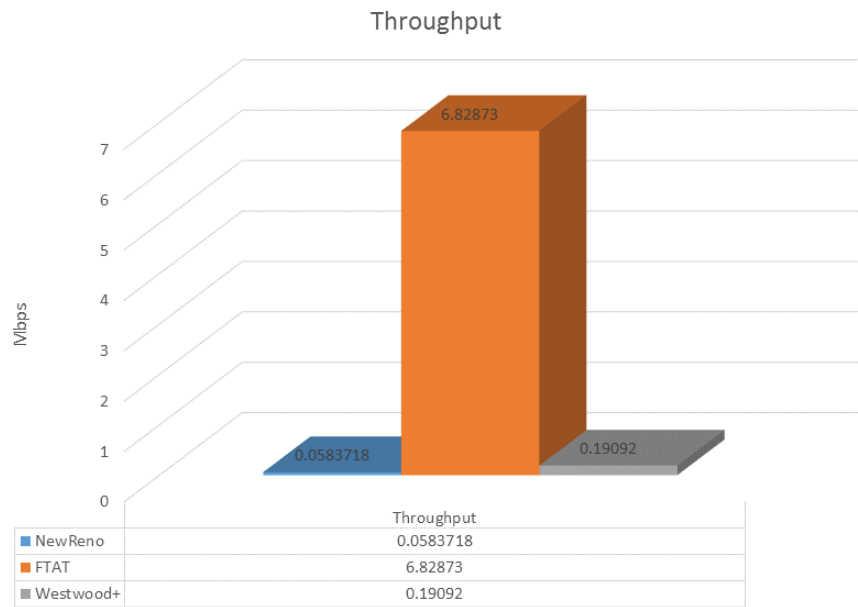


Figure 30: Throughput and goodput topology four

In this scenario, a connection with 10 Mbps, and two competing users sending and receiving data concurrently over the connection. Figure 30 shows that when the two TCP users was using NewReno as the congestion control algorithm, each node achieved about

0.0577 Mbps, and this why NewReno is not a suitable congestion control in high bandwidth-delay product networks. On the other hand, Westwood was able to achieve throughput improvement over NewReno due to the adaptive mechanism used by Westwood, but still was unable to achieve a proper share of the available network bandwidth, and it achieved about 0.189 Mbps. In case of FTAT, the algorithm was able to detect most of the false congestion signals due to the testing mechanism, and as a result was able to utilize the available bandwidth, and it achieved about 6.45 Mbps.

CHAPTER VIII

Evaluation and Comparison of Different Congestion Control Algorithms of Linux Stack Using DCE Cradle (Direct Code Execution Cradle)

8.1.Introduction

Simulation based comparison is helpful for any new protocol or addition to an existing one. Although the benefits of using simulators means reducing the development and testing time, reproducibility, cost effectiveness, etc., there are limitations of using simulation. For example, protocols implementations are not available for all the algorithms in every simulator. For these reasons and others, researchers do not rely only on the simulations, but they use it as an important indication toward the performance of a protocol. Another alternative is to use emulators, which combine the benefits of simulations and real systems. Emulators, however, also have drawbacks such as they are mostly based on Virtual Machines and run on real-time, which makes them limited to the capabilities of the hardware. As well, debugging an emulated system is very difficult.

Another solution is the use of real stacks of operating systems in simulators such as ns-3 which is open source discrete-event widely deployed and accepted in the networking community, for realistic results. Currently the widely deployed ways of using real kernel stacks in simulators is through Network Simulation Cradle (NSC) [37], Direct Code

Execution (DCE) [38], and Direct Code Execution Cradle (DCE Cradle) [39]. The use of such frameworks has been shown in many research papers that they produce very accurate results.

Our choice here was to use DCE Cradle, since it is an open source framework that provides more features than DCE by allowing utilization of ns-3 applications, and provides more Linux kernel versions support than NSC.

8.2.Simulation and Comparisons

8.2.1. Topology One: One-way Geo Satellite Scenario

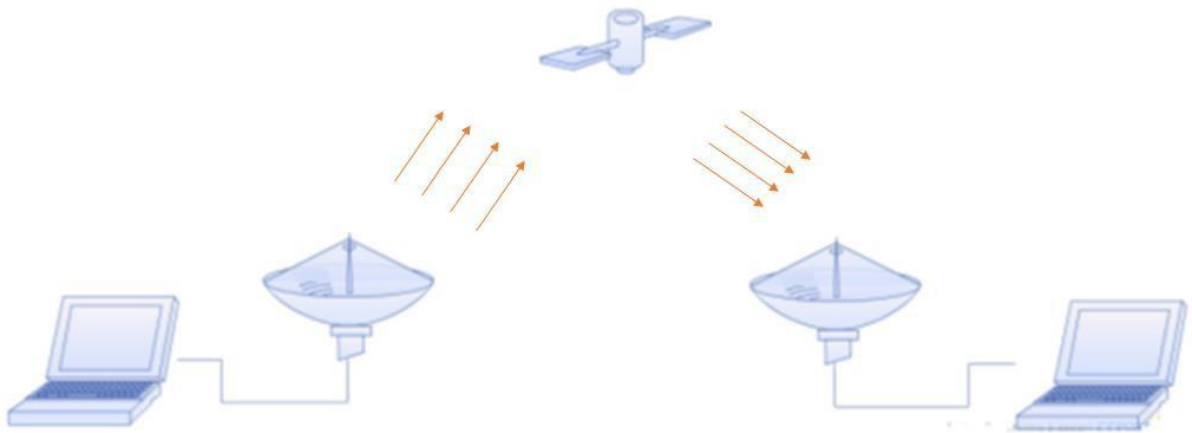
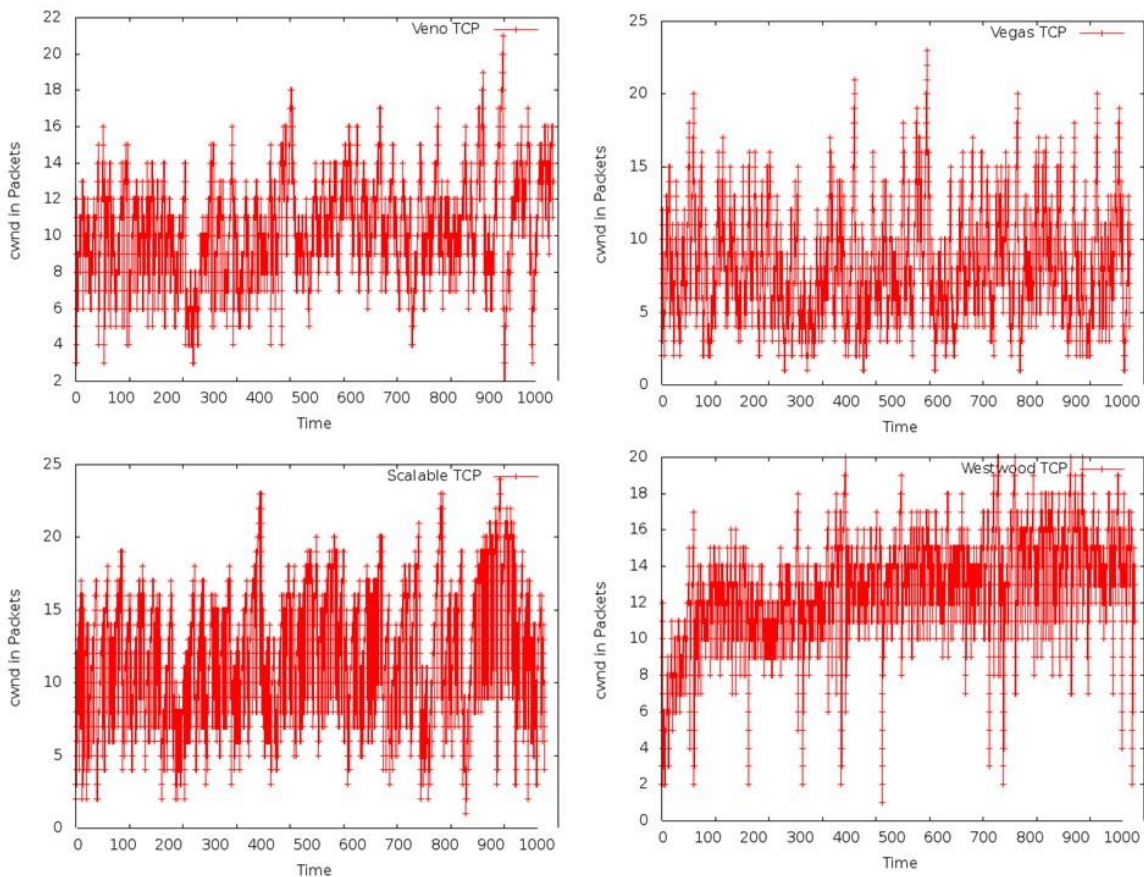


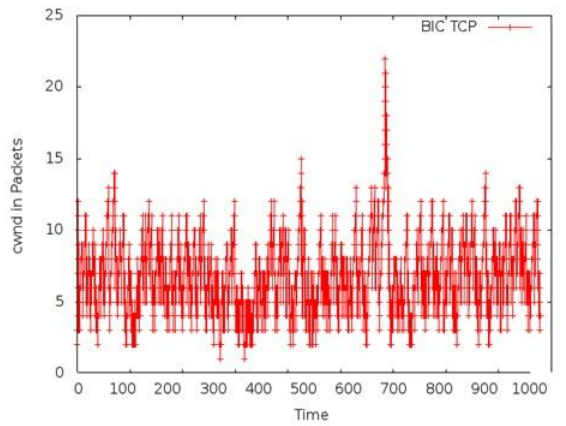
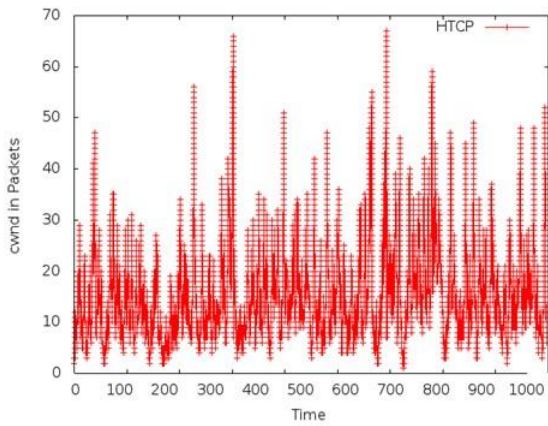
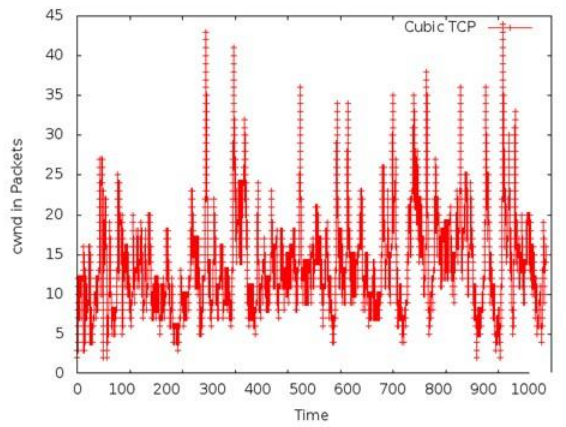
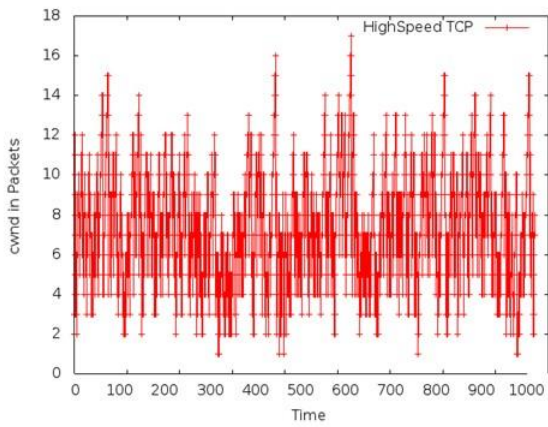
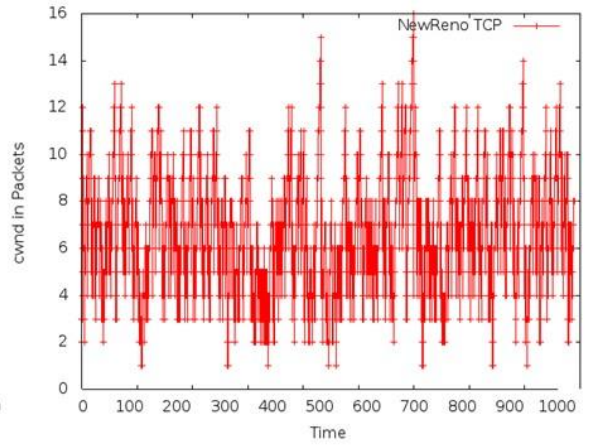
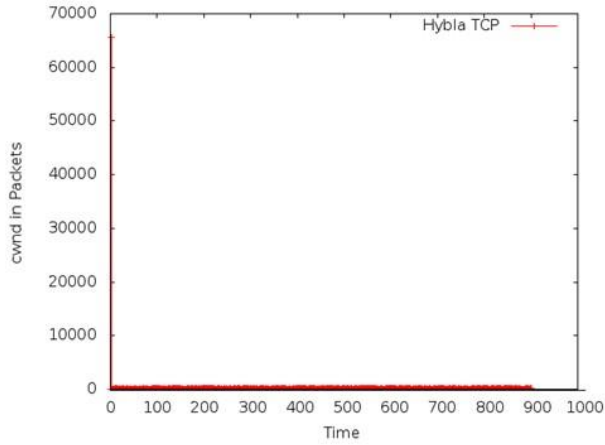
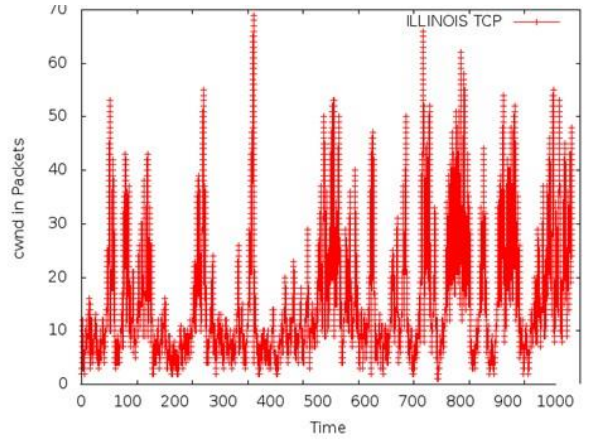
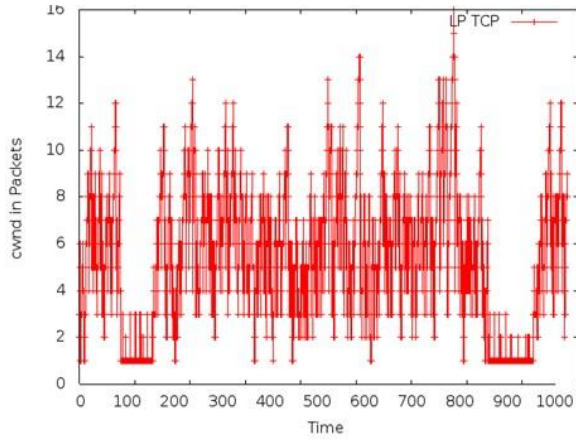
Figure 31: Topology one.

This topology (Figure 31) is helpful in showing the behavior of the congestion control protocol in high BDP networks with loss rate. It is also important to note that there is a difference between this scenario and the one used with ns-3, as the flow here is

only one-way directional, as opposed to the two-way flow in that case (Section 6.4). The node on the left side uses the different TCP-variants, and the node on the right side is TCP NewReno. The link exhibits a loss rate of 2% with a long delay of 270 ms. The connection bandwidth is 10 Mbps, the simulation time is 1000 seconds, and the packet size is 536 bytes.

There are 14 scenarios for this topology, which examine every congestion control implementation in the Linux Stack. The Congestion control algorithms that are examined in this scenario are Reno, BIC, Cubic, Westwood, Highspeed, Hybla, Vegas, Veno, HTCP, Scalable, LP, YeAH, Illinois, and FTAT.





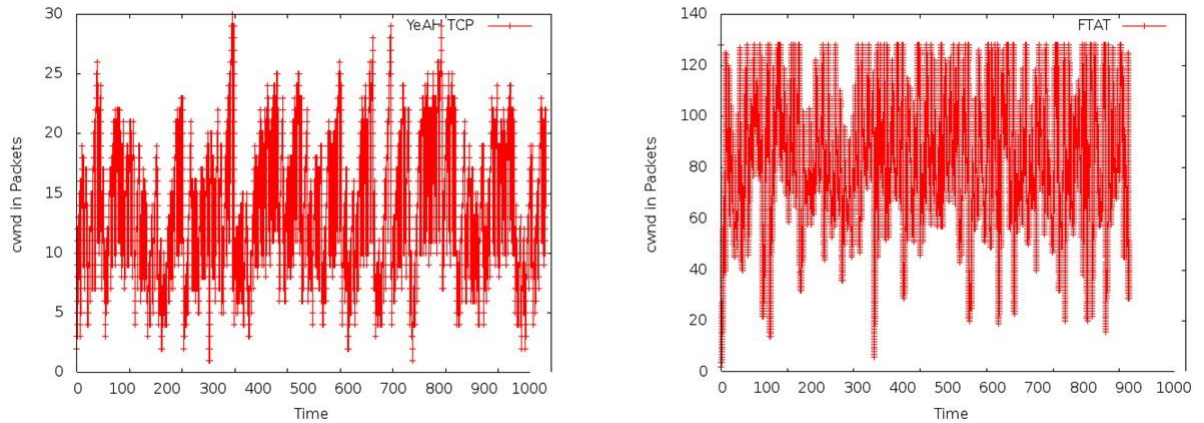


Figure 32: cwnd topology one

It should be noted here that DCE does not have a framework such as in ns-3 for tracing capabilities, which requires individual efforts to implement the traces. The *cwnd* tracing included much more points than that provided in ns-3 graphs which makes the graphs not very smooth. Also due to the differences in the implementation of the Linux for TCP and ns-3 for TCP, the *cwnd* graphs have different scales. Linux *cwnd* is implemented in terms of number of packets, while ns-3 implement the *cwnd* in terms of bytes.

The *cwnd* for each algorithm is shown in Figure 32. It can be observed that each algorithm has upper limit for the *cwnd*, Cubic reached about 45 packets, NewReno reached about 16 packets, Veno around 21 packets, Vegas around 24 packets, Westwood 20 packets, FTAT 130, and so on. Hybla *cwnd* reached around 70,000 packets, which is out of range and impractical, but this spark can be explained as a bug in the implementation of the algorithm in the Linux kernel.

The graph in Figure 33 shows the sequence number advancement gathered in one graph. Due to the huge number of points from the traces, it was not possible to make the graph, as a result a number of points was chosen to implement the graph and

show the relative advancement of the sequence number for each algorithm grouped in single graph.

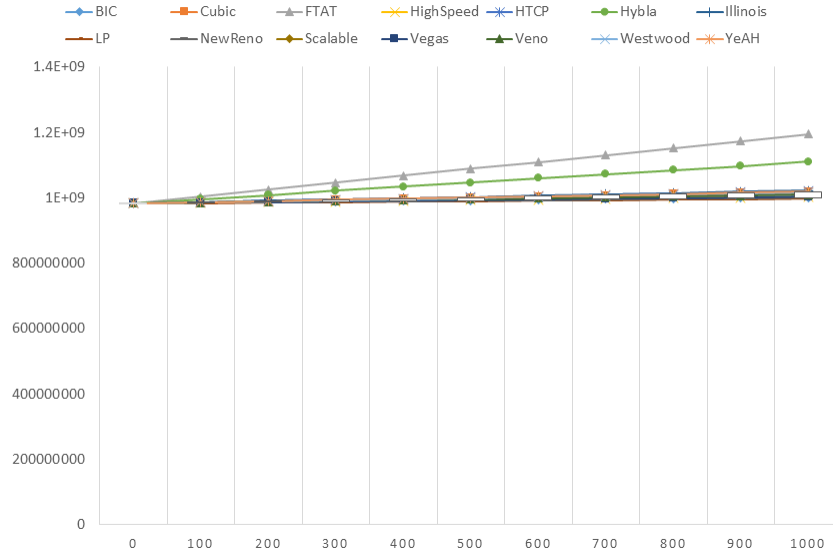


Figure 33: Sequence number topology one

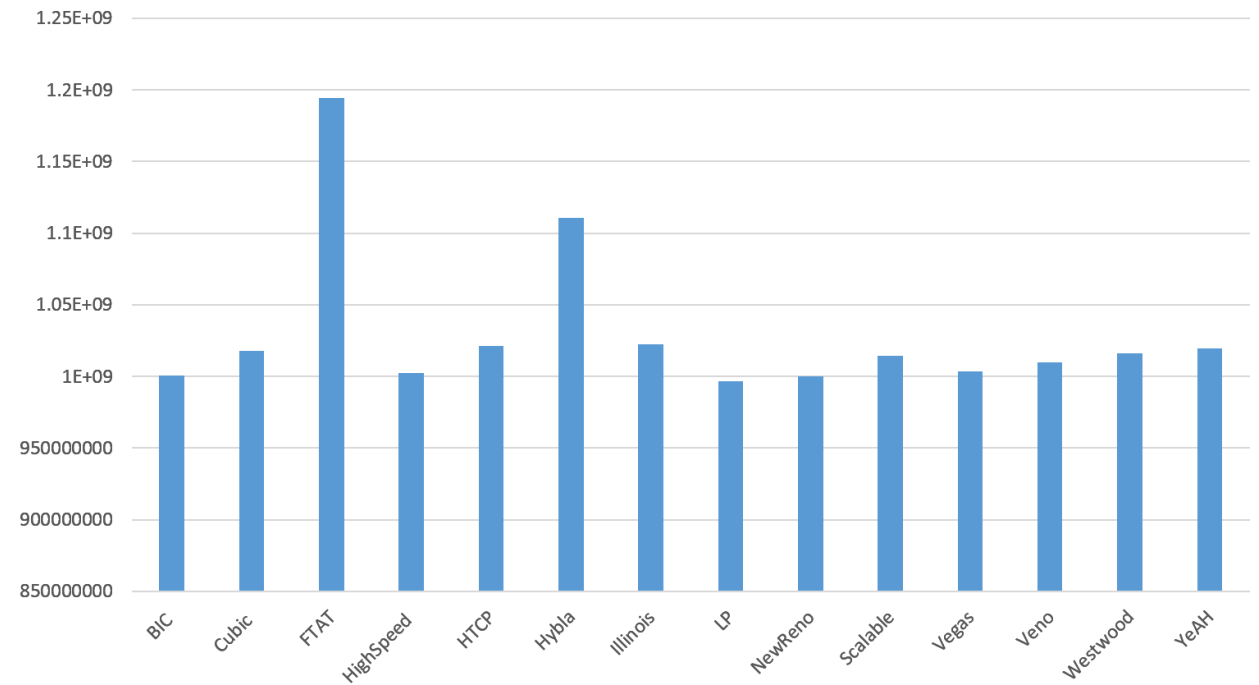


Figure 34: Highest sequence number

In Figure 34, it's shown the highest sequence number achieved by each congestion control. FTAT achieved the highest sequence number, followed by Hybla, and then most of the algorithms varies slightly.

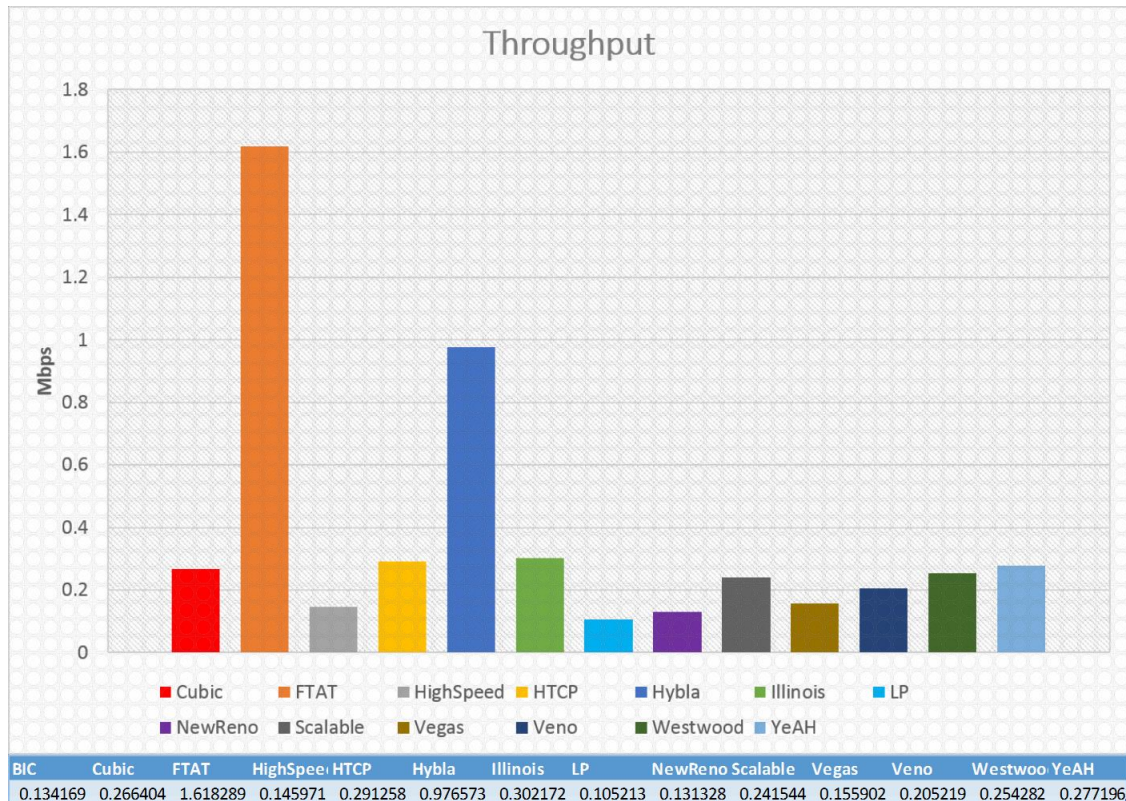


Figure 35: Throughput topology one

In Figure 35, the throughput measurements is presented. FTAT algorithm achieved about 1.6 Mbps, while Hybla achieved about 1 Mbps, and Cubic achieved about 0.266 Mbps. It is proper to emphasize here that NewReno and Westwood+, gained throughput improvement over the topology in Section 6.4. This is due to the nature of this connection, which is one-way that is no competing user on the same connection. While FTAT experienced throughput degradation, this can be a result of the experimental implementation of FTAT in linux and the changes in the topology.

8.2.2. Topology Two

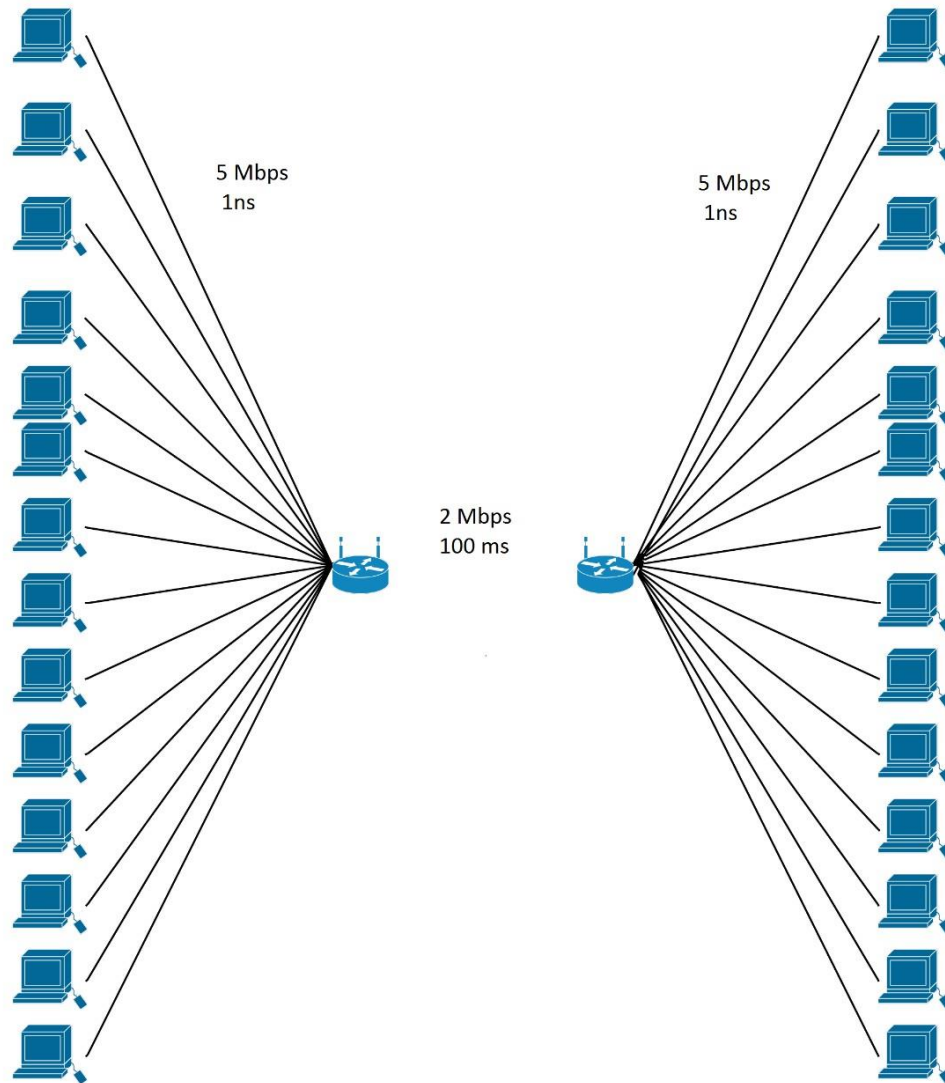


Figure 36: Topology two

The second topology (Figure 36) gathers the 14 algorithms in one network in order to get more insight about the behavior of each congestion control algorithm when competing with other nodes. The bottleneck link is the link between the two routers,

which is a wireless link and has a packet loss rate of 2%, bandwidth of 2Mbps, and a delay of 100 ms. The access links are 5Mbps each with a delay of 1 ns.

The flows start at 4 seconds, and the simulated time is 1000 seconds, the throughput is shown in Figure 37.

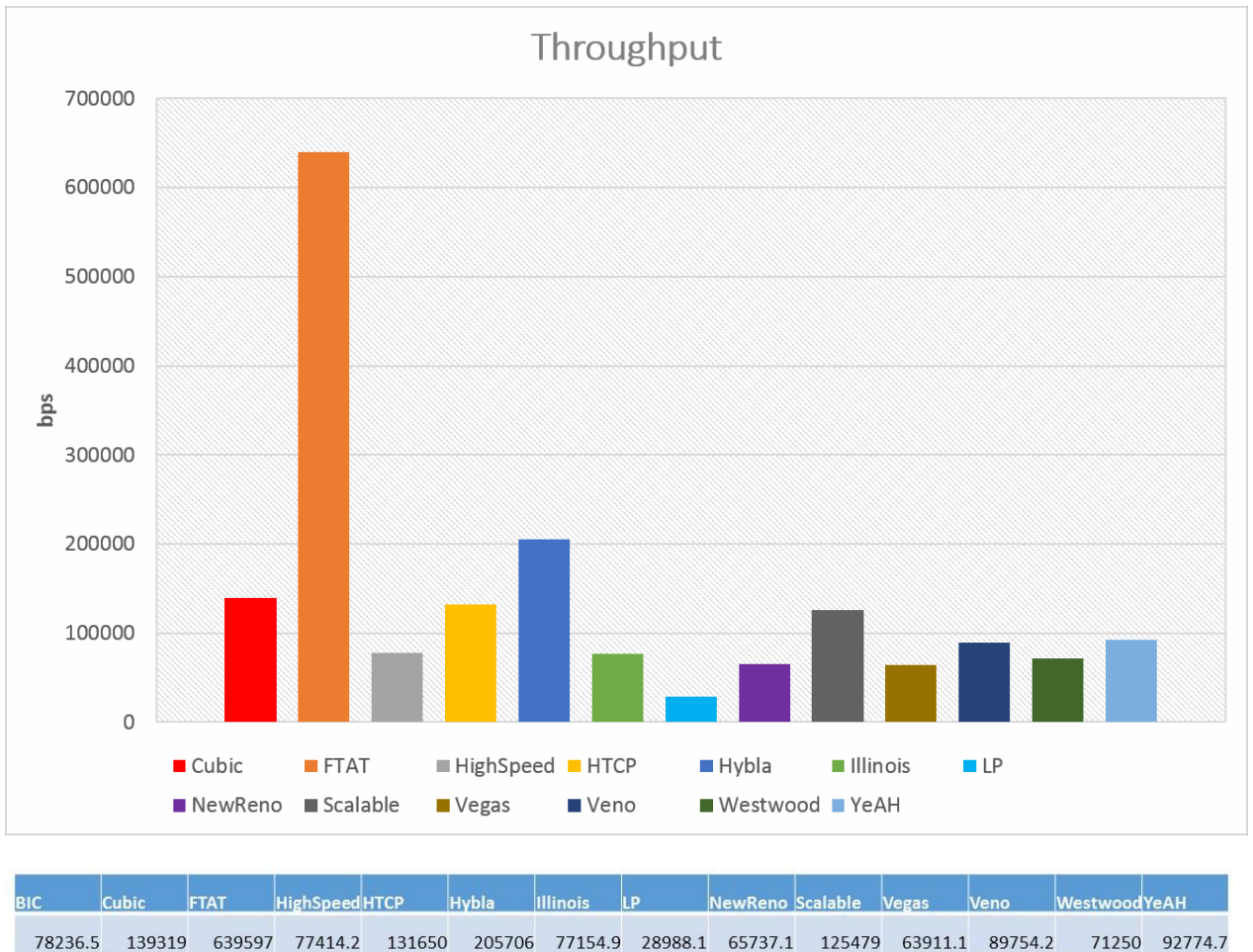


Figure 37: Throughput topology two

Again here in Figure 37, it is shown that FTAT algorithm achieved the best throughput of about 640 Kbps, while Hybla of about 206 Kbps, and Cubic of about 139 Kbps. With the high volume of traffic on the bottleneck link that is characterized by the long propagation delay, most of the nodes controlled by a specific TCP congestion control fall under 100 Kbps.

8.2.3. Topology Three

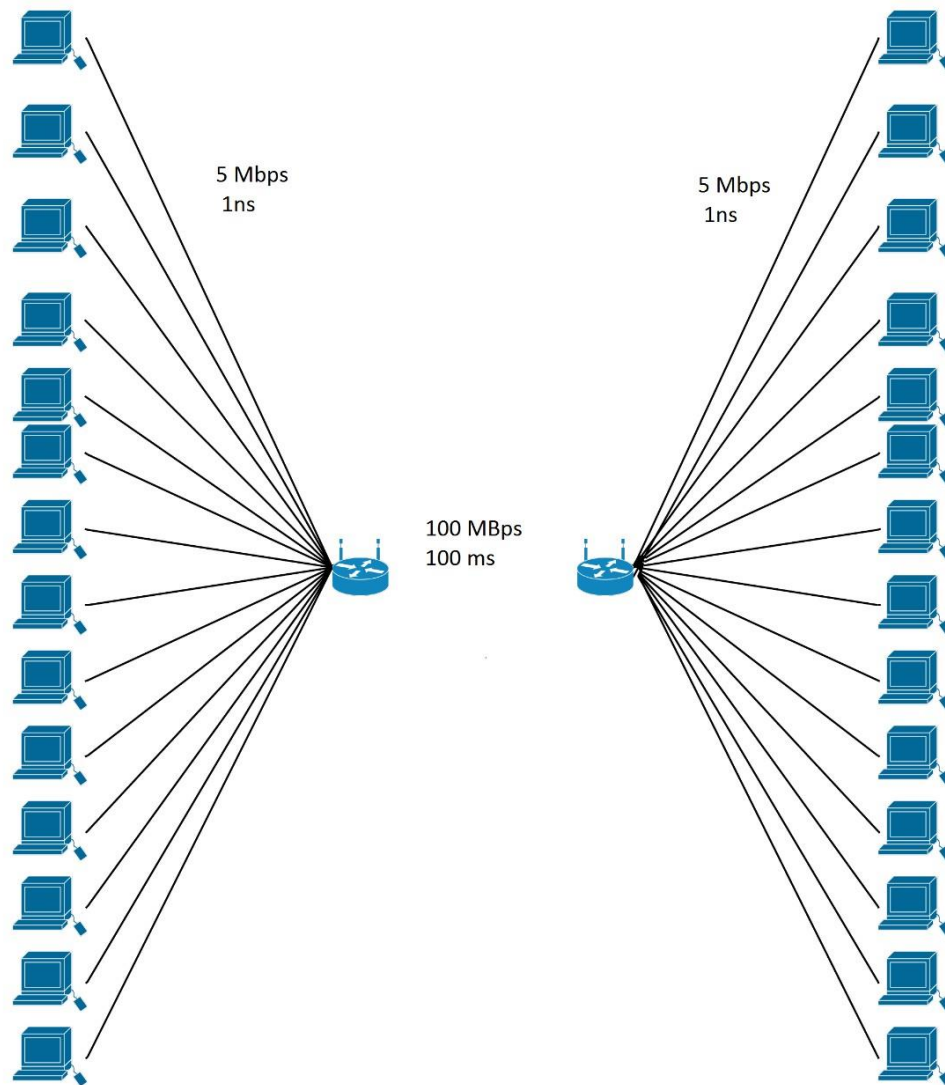


Figure 38: Topology Three

To further investigate the effect of high BDP links, we set the same topology with the same conditions (Figure 38), except the bottleneck link changed to 100 Mbps. The throughput is shown in Figure 39.

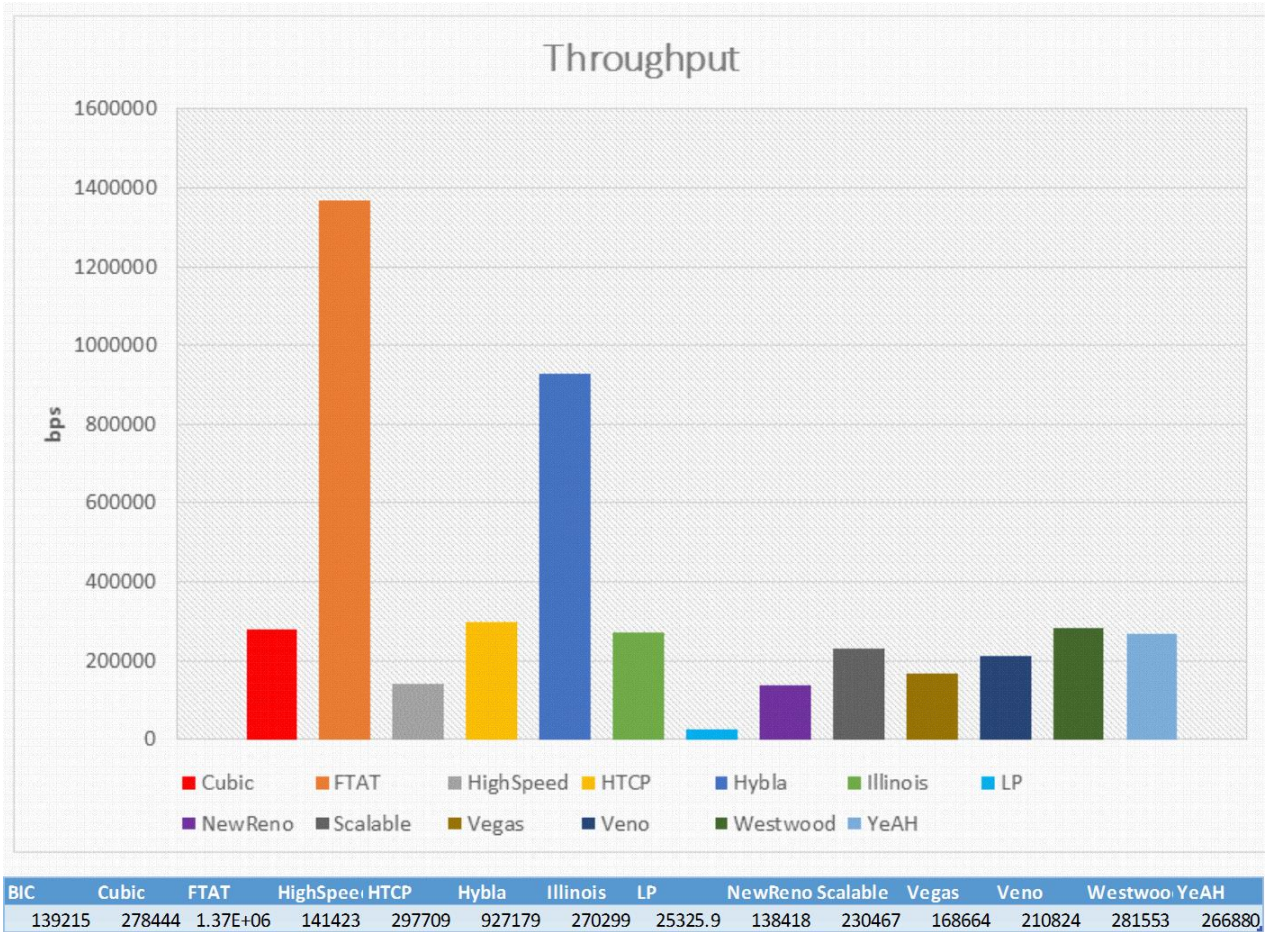


Figure 39: Throughput topology three

For the sake of experimenting the different congestion control algorithms in linux, the bandwidth of the bottleneck link was increased. In Figure 39, it is shown that FTAT achieved the best throughput of about 1.4 Mbps, while Hybla of about 1 Mbps, and Cubic of about 278 Kbps. Many of the congestion control in linux was still unable to increase the throughput to a proper values in the presence of the long delay.

8.2.4. Fourth Topology

The last topology (Figure 40) examines the effect of high bandwidth, low delay networks. The settings are the same as the previous topology, except for the wireless link between the two routers, which is changed to 1 ms.

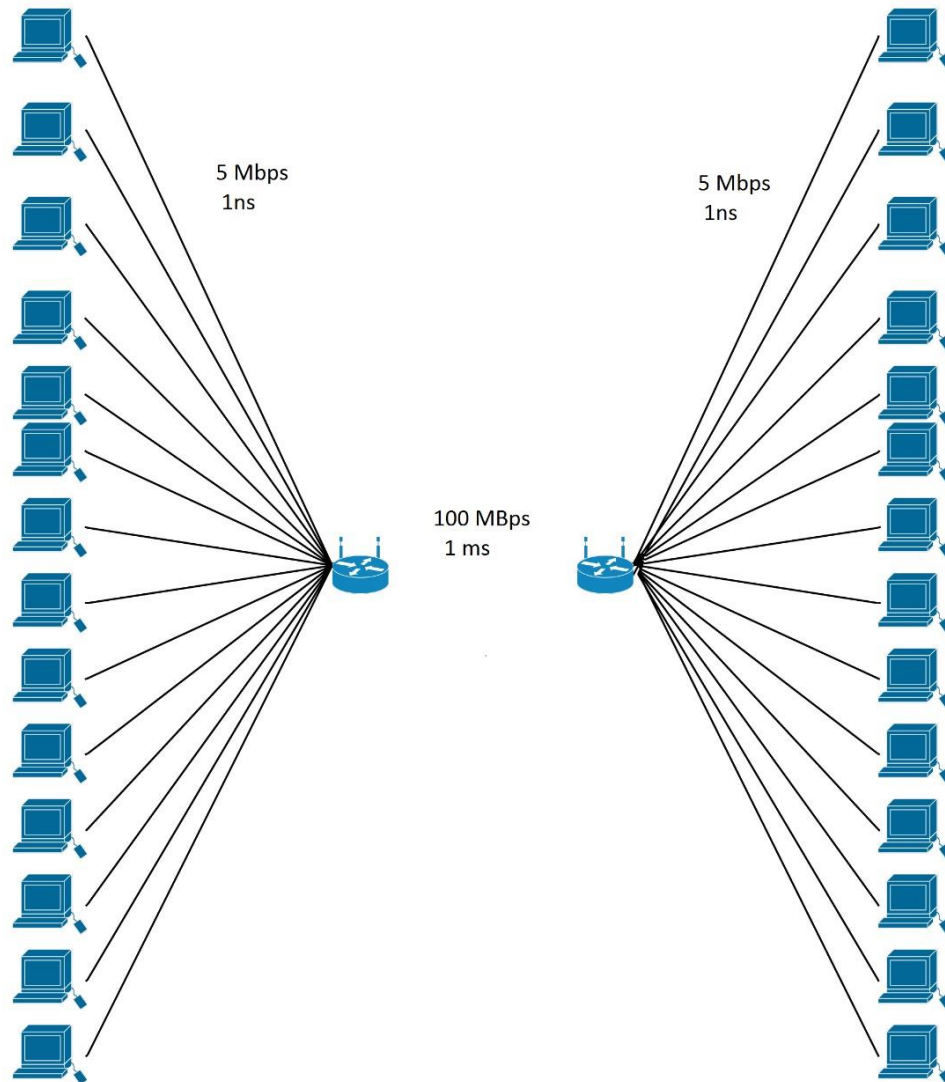
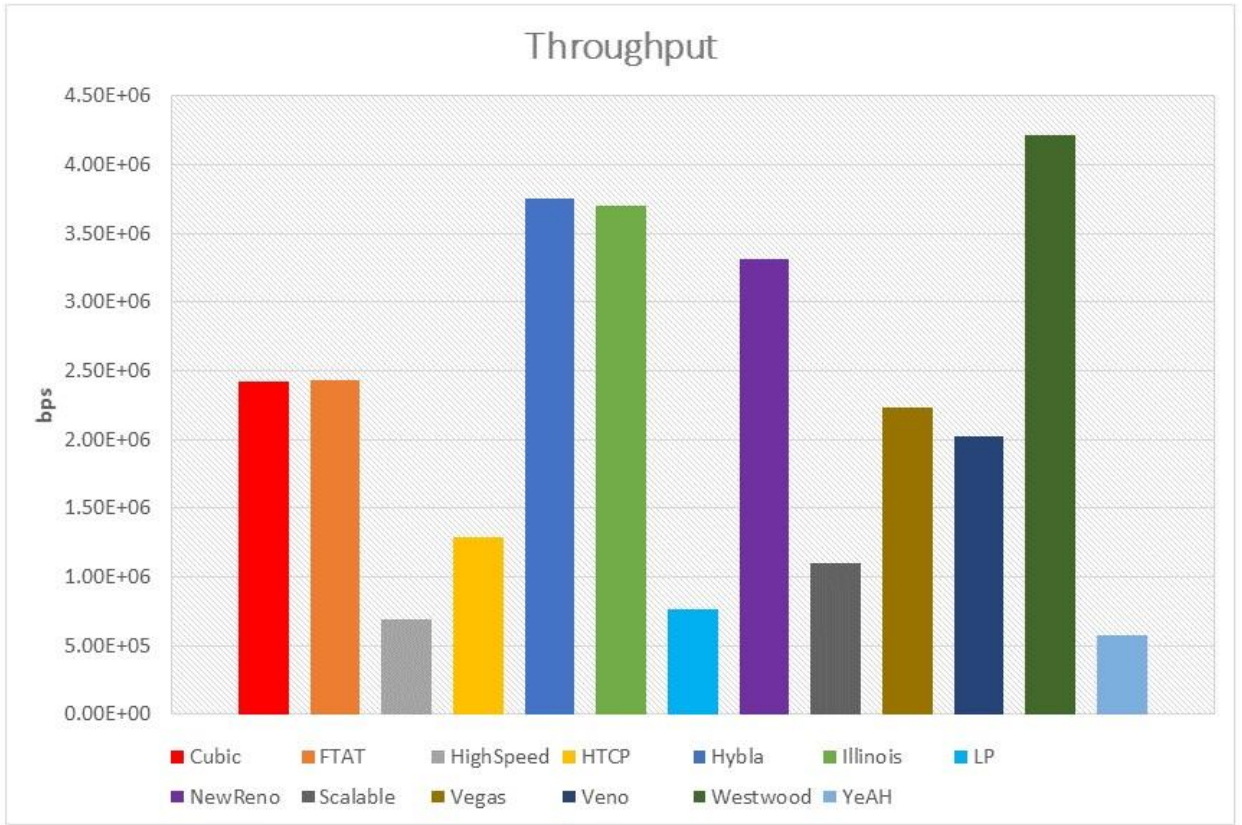


Figure 40: Topology Four

The throughput analysis is shown in the next graph.



BIC	Cubic	FTAT	HighSpeed	HTCP	Hybla	Illinois	LP	NewReno	Scalable	Vegas	Veno	Westwoo	YeAH
4.09E+06	2.42E+06	2.44E+06	694267	1.28E+06	3.75E+06	3.70E+06	760358	3.32E+06	1.10E+06	2.23E+06	2.02E+06	4.22E+06	576497

Figure 41: Throughput topology four

In Figure 41, it is shown great improvement of the throughput for the different congestion control algorithms in linux TCP stack. Westwood was the most algorithm that benefited from the propagation delay reduction, which shows that it is greatly dependent on the propagation delay of the links in a network. NewReno algorithm also is one of the highest algorithms in the graph, which emphasize again that it was designed for networks with specific characteristics. Cubic also shows big improvement in the throughput in comparison to the results of the topology in the last section. FTAT achieved throughput improvement, but was unable to achieve the best throughput, as the way FTAT operates

require relatively high propagation delays, which makes it excellent for the networks with high bandwidth-delay product. Although FTAT did not achieve the best results in this topology, it still ranks as one of the best algorithms operated in this network, with slight throughput improvement over Cubic TCP congestion control.

CHAPTER IX

Conclusion

The proposed algorithm has been tested through simulation using ns-3, and the Linux kernel code against the major deployed congestion control algorithms. It was shown in the simulation results that FTAT gains a throughput improvement over other TCP congestion control algorithms in wireless and high bandwidth-delay product networks.

The steady state throughput mathematical model derived for FTAT shows that FTAT is stable even under severe congestion conditions, which was supported by the experiments conducted using simulations in ns-3 and DCE - Linux that FTAT throughput is limited to the network available bandwidth. The fundamental principle which makes FTAT is stable that FTAT uses self-clocking as NewReno to send new data to the network but differ in the way FTAT interpret congestion state.

Although there are indications through the experiments conducted that FTAT is friendly to other TCP congestion control algorithms and has Fairness among connections operating with FTAT. Fairness and Friendliness is a case study, and it can be completed in future work.

From our study and the experiments that was conducted, we have found that due to the additive-increase multiplicative-decrease of NewReno, it is unable to grow the congestion window to a proper sizes in high bandwidth and long delays networks, and even after long period of time in a connection, a single packet time-out will reduce the

congestion window to 1 segment, and three duplicate acknowledgements will reduce the congestion window to half of its current value. On the other hand FTAT is Adaptive-Increase Adaptive-Decrease, which measures the network's available bandwidth upon three duplicate acknowledgements, and adjusts the congestion window accordingly. Also from the experiments that have been conducted, it was observed that FTAT does not take from the bandwidth of the other algorithms sharing the network. Instead, it achieves the network available bandwidth.

The current implementations of the congestion control such as NewReno, gives stability for a TCP connection and prevents congestion in the network, but in the same time it limits the throughput and results in wasted bandwidth in the case of high BDP networks. Although NewReno was considered as the standard congestion control for TCP for many years, it is unable to grab the available bandwidth in a connection in long delay networks, due to its slow rate of the *cwnd* increase. Most of the congestion control implementations in TCP follow the same rule, which are not suited for today's high-speed networks.

FTAT offers an adaptive mechanism that allows the window to grow when such growth is possible, and to be conservative when the available bandwidth is limited. FTAT is best described as using an Adaptive-Increase Adaptive-Decrease paradigm. It gets the best results when it is operated in networks with high bandwidth in the presence of large delays, and in wireless networks where there random packet loss. FTAT does not rely on the high bandwidth in order to perform well, but it requires a loss rate or a delay in the network in order for the adaptive mechanism to have the optimum results.

In order for FTAT to gain its working mechanism, the sending and receiving buffers size should be set to a value large enough in order to buffer the packets not in ordered, in the

adaptive state. To the best of our knowledge, there are no constraint on the buffer sizes, and it is adjustable in every Linux system. The value that was set in the simulations and experiments in this thesis was 5 Megabytes, which does not introduce any overhead. FTAT currently has experimental implementations in the Linux kernel stack (3.2.0 and 2.6.36), and in ns-3. As a future work, we will submit the implementations to be officially part of the ns-3 and Linux OS.

BIBLIOGRAPHY

- [1] V. Jacobson “Congestion Avoidance and Control”, Proceeding SIGCOMM '88 Symposium proceedings on Communications architectures and protocols, pp. 314 – 329, 1988.
- [2] Floyd, S., Henderson, T. New Reno Modification to TCP's Fast Recovery, RFC 2582, April 1999.
- [3] Jacobson, V., "Berkeley TCP Evolution from 4.3-Tahoe to 4.3, Reno," Proceedings of the 18th Internet Engineering Task Force, University of British Columbia, Vancouver, BC, September 1990.
- [4] M. Allman, S. Floyd, C. Partridge “Increasing TCP’s Initial Window” RFC 3390, October. 2002.
- [5] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, Ren Wang. “TCP Westwood: End-to-End Congestion Control for Wired/Wireless Networks”, Wireless
- [6] Luigi A. Grieco and Saverio Mascolo. “TCP Westwood and Easy RED to Improve Fairness in High-Speed Networks”, Seventh International Workshop on Protocols For High-Speed Networks (PfHSN’2002), Berlin, Germany, April. 2002.
- [7] Lawrence S. Brakmo, Sean W. O’Malley, Larry L. Peterson. “TCP Vegas: New Techniques for Congestion Detection and Avoidance”, Proc. ACM SIGCOMM ’94, Vol. 24 Issue 4, Pages 24-35, Oct. 1994.
- [8] Jong Suk Ahn, Peter B. Danzang, Zhen Liu, Limin Yan. “Evaluation of TCP Vegas: Emulation and Experiment”, Proc. ACM SIGCOMM 95, Vol. 25 Issue 4, Pages 185-195, October 1995.

- [9] J. Mo et al., “Analysis and comparison of TCP/Reno and Vegas,” in Proc. IEEE INFOCOM, pp. 1556–1563, 1999.
- [10] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, Randy H. Katz, “Improving TCP/IP performance over wireless networks”, Proceeding MobiCom '95 Proceedings of the 1st annual international conference on Mobile computing and networkin, PP. 2 – 11, 1995.
- [11] K. Brown and S. Singh, “M-TCP: TCP for Mobile Cellular Networks”, ACM Computer Communications Review, vol. 27, no. 5, pp. 19-43, 1997.
- [12] San-Qi Li, Song Chong, Chia-Lin Hwang “Link capacity allocation and network control by filtered input rate in high speed networks”, IEEE/ ACM, Networking Transactions , vol. 3, no. 1, pp. 10 – 25, 1995.
- [13] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz, “Improving TCP/IP Performance over Wireless Networks”, Proceedings of the 1st ACM Conference on Mobile Computing and Networking, Berkeley, CA, November 1995.
- [14] Frank Kelly. “Mathematical modelling of the Internet”, Proceedings of the Fourth International Congress on Industrial and Applied Mathematics, PP. 105 – 116, 1999.
- [15] Luigi A. Grieco and SaverioMascolo. “Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control”, IFIP International Federation for Information Processing, 2009.
- [16] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan and Randy H. Katz “A Comparison of Mechanisms for Improving TCP Performance over Wireless Links”, Networking, IEEE/ACM Transactions on, (Volume 5, Issue: 6), pp. 756 – 769, Dec 1997.

- [17] James F. Kurose, Keith W. Ross, Computer Networking: A Top-Down Approach, 5/E, ©2010 • Addison-Wesley • Cloth, 864 pp.
- [18] Grieco, L. A and Mascolo, S., End-to-End Bandwidth Estimation for Congestion Control in Packet Networks. Second International Workshop, QoS-IP 2003, Milano, Italy, February 2003.
- [19] Xylomenos, G., Polyzos, G.C., Mahonen, P., Saaranen, M., “TCP Performance Issues over Wireless Links”, Communications Magazine, IEEE (Volume:39 , Issue: 4), pp. 52 – 58, Apr 2001.
- [20] K. Ramakrishnan and S. Floyd, “A Proposal to add Explicit Congestion Notification (ECN) to IP”, RFC 2481, January 1999.
- [21] Shohei Kodama, Masayoshi Shimamura, Katsuyoshi Iida. “Initial CWND Determination Method for Fast Startup TCP Algorithms”, Quality of Service (IWQoS), 2011 IEEE 19th International Workshop, pp. 1 – 3, 2011.
- [22] http://nslam.isi.edu/nslam/index.php/User_Information.
- [23] Thomas R. Henderson, Mathieu Lacage, George F. Riley. “Network Simulations with the ns-3 Simulator”, Proc. ACM SIGCOMM, 2008.
- [24] GJAM Carneiro. “NS-3: Network Simulator 3”, <https://www.nslam.org/tutorials/NS-3-LABMEETING-1.pdf>, April 2010.
- [25] M. Allman, V. Paxson, and W. Stevens, “TCP congestion control,” IETF, RFC 2581, Apr. 1999.
- [26] Jacobson, V., "Berkeley TCP Evolution from 4.3-Tahoe to 4.3, Reno," Proceedings of the 18th Internet Engineering Task Force, University of British Columbia, Vancouver, BC, September 1990.

- [27] Lisong Xu, Khaled Harfoush, and Injong Rhee. “Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks”, INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, PP. 2514 - 2524 vol.4, March 2004.
- [28] Sangtae Ha, Injong Rhee, Lisong Xu. “CUBIC: A New TCP-Friendly High-Speed TCP Variant”, ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel, PP. 64-74, vol. 42, July 2008.
- [29] Robert Morris. “Scalable TCP Congestion Control”, Proceedings of the IEEE INFOCOM 2000 Conference, PP. 1176 – 1183, vol. 3, Mar 2000.
- [30] S. Floyd. “HighSpeed TCP for Large Congestion Windows”, RFC 3649, December 2003.
- [31] D. Leith, R. Shorten. “H-TCP: TCP for high-speed and long-distance networks”, Proceedings of PFLDnet, 2004.
- [32] Shao Liu, Tamer Basar, R. Srikant. “TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks”, Performance Evaluation, PP. 417–440, vol. 65, June 2008.
- [33] Carlo Cainin, and Rosario Firrincieli. “TCP Hybla: a TCP enhancement for heterogeneous networks”, International Journal of Satellite Communications and Networking, PP. 547 – 566, vol. 22, October 2004.
- [34] Cheng Peng Fu, and Liew, S.C. “TCP Veno: TCP enhancement for transmission over wireless access networks”, PP. 216 – 228, vol. 21, February 2003.
- [35] A Kuzmanovic, EW Knightly. “TCP-LP: low-priority service via end-point congestion control”, Networking, IEEE/ACM Transactions, PP. 739 – 752, vol. 14, August 2006.

[36] Andrea Baiocchi, Angelo P. Castellani and Francesco Vacirca. “YeAH-TCP: Yet Another Highspeed TCP”, Proceedings of PFLDnet, 2007.

[37] Sam Jansen and Anthony McGregor. “Simulation with Real World Network Stacks”, IEEE - Simulation Conference, December 2005.

[38] Hajime Tazaki, Frédéric Urbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turetletti, Walid Dabbous. “Direct code execution: revisiting library OS architecture for reproducible network experiments”, Proceedings of the ninth ACM conference on Emerging networking experiments and technologies, PP. 217 – 228, 2013.

[39] Hajime Tazaki, Frédéric Urbani, Thierry Turetletti. “DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism”, Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, PP. 153 – 158, 2013.